

Serial #

17568

The Time Saver



ARCHIMEDES
SOFTWARE

Archimedes
C-6811 Cross-Compiler Kit
For
68HC11 Microcontroller Development

Third Edition
April, 1992

Copyright (c) 1987, 1992 Archimedes Software, Inc.

Archimedes Software, Inc., San Francisco, CA.

SINGLE-CPU LICENSE AGREEMENT

BY USING THIS SOFTWARE YOU HAVE AGREED TO THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT.

The Licensed "Program" (Software and Documentation) is owned and copyrighted by Archimedes Software, Inc. with all rights reserved. The Program is licensed to You only under the terms and conditions set forth below:

YOU MAY only use this Program at one CPU at a time (PC, VAX or MicroVAX). You may only make copies of this Program for back-up and archival purposes only. You may only incorporate the libraries into any derivative products.

YOU MAY NOT make copies of the Program except as set forth above. You may not incorporate any parts of this Program, except as set forth above, into any other products. You may not lease, rent, grant sub-licenses, transfer or sell the Program to any other party. You may not use the Program in a computer service business or time-sharing business. You may not use the Program on a network unless all CPUs are each licensed by Archimedes Software, Inc.. You may not make any alterations in the Program.

WARRANTY Archimedes Software, Inc. warrants that the physical media, on which the software is shipped, is free from defects in material and workmanship for a period of thirty days from date of delivery to You as evidenced by receipt. Your exclusive remedy for breach of this Warranty shall be the replacement copy of the Program.

ARCHIMEDES SOFTWARE, INC. PROVIDES THE LICENSED PROGRAM "AS IS" WITHOUT ANY OTHER WARRANTIES INCLUDING BY NOT LIMITED TO WARRANTIES FOR MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE. ARCHIMEDES SOFTWARE, INC. SHALL NOT BE LIABLE FOR ANY DAMAGES, INCLUDING BUT NOT LIMITED TO INTERRUPTION OF BUSINESS, LOSS OF PROFIT, INCIDENTAL, CONSEQUENTIAL OR ANY OTHER CLAIMS.

MONEY-BACK GUARANTEE PC-hosted software comes with a 30 day money back guarantee, described on a separate page.

TERMS Archimedes Software, Inc. reserves the right to terminate this Agreement if you are in breach with any term and condition set forth in the Agreement.

GOVERNING LAW This Agreement shall be construed and governed by the laws of the state of California and the U.S. Federal Government.

Disclaimer

The information in this document is subject to change without notice. While the information contained herein is assumed to be accurate, Archimedes Software, Inc. assumes no responsibility for any errors or omissions.

Copyright Notice

Copyright (c) 1987, 1992 Archimedes Software, Inc., CA, United States. No part of this document may be reproduced without the prior written consent of Archimedes Software, Inc.. The software described in this document may only be used as a licensed product in accordance with the terms and conditions of an Archimedes Software License Agreement.

Trademarks

Archimedes and Microcontroller C are trademarks of Archimedes Software, Inc.
MS-DOS is a trademark of Microsoft Corp.
UNIX is a trademark of AT&T and Bell Labs.
VAX, MicroVAX, VMS and Ultrix are trademarks of Digital Equipment Corporation.

Edition

Third Edition, April 1992.

Table Of Contents

PREFACE.....	x
---------------------	----------

TUTORIAL.....	T-1
----------------------	------------

T.1 What this Tutorial Covers	T-1
T.2 Standard Files and File Types.....	T-2
T.3 Installation.....	T-3
T.4 Other Development Tools	T-3
T.5 Archimedes C and the 6811	T-4
T.6 The Development System	T-7
T.7 Putting Together the Example Program.....	T-10
T.7.1 Modifying and Assembling CSTARTUP	T-11
T.7.2 The PUTCHAR and GETCHAR Routines	T-15
T.7.3 The EXAMPLE.C Program.....	T-17
T.7.4 Compiling the Program	T-20
T.7.5 An Assembly Function.....	T-23
T.7.6 Linking the Example Program.....	T-24
T.7.7 Downloading and Testing.....	T-29

I. C-6811 COMPILER.....	I-1
--------------------------------	------------

1. Overview.....	I-1
1.1 Introduction.....	I-1
1.2 Chapter Overview	I-1
1.3 C-6811 Overview.....	I-2
2. Data Representation.....	I-3
3. Memory Models	I-5
4. Absolute Read/Write at C-level.....	I-10
5. Register Usage.....	I-11
6. Assembly Language Interface.....	I-11
7. Parameter Stack	I-12
8. Linking.....	I-14
9. Configuration Issues.....	I-18
10. Stack and Heap Size	I-19
11. Interrupt Routines	I-20

12.	68HC11-Specific Extensions.....	I-21
12.1	In-line Functions	I-21
12.2	Extended Language Keywords.....	I-22
12.3	Additional Examples.....	I-31
13.	Operating Instructions.....	I-35
14.	Files.....	I-35
15.	Compiler Switches and Options	I-36
16.	Include Files.....	I-46
17.	Compiler Diagnostics.....	I-47
18.	C-68HC11 Compatibility.....	I-48
18.1	New ANSI C keywords	I-49
18.2	Additional ANSI Adaptations.....	I-52
18.3	Additional Language Extensions.....	I-53
18.4	Pre-processor Directives	I-54
19.	C-Compiler Extensions.....	I-55
20.	Sample Code.....	I-56
II.	ASSEMBLER	II-1
1.	Assembler Overview.....	II-1
1.2	Characteristics and Features	II-2
1.2.1	Basic Features	II-2
1.2.2	68HC11 Chip Support.....	II-2
1.2.3	Compatibility	II-3
2.	Using the Assembler	II-4
2.1	Installation.....	II-4
2.2	File Naming Conventions	II-5
2.3	Editing Source Files.....	II-5
2.4	Basic Operation.....	II-6
2.4.1	Overview.....	II-6
2.4.2	Simple Command line Examples	II-6
2.4.3	Linking the Demo Program	II-7
2.5	Command line Operation.....	II-9
2.5.1	Command line Syntax	II-9
2.5.2	Command line Parameters and Options	II-10
2.5.3	Command line Examples.....	II-13
2.6	Prompted Operation.....	II-14
2.7	Error Messages.....	II-15
2.8	Linking A6801 Programs	II-16
2.8.1	Overview.....	II-16
2.8.2	XLINK Command Lines	II-17

2.8.3	Command line Examples.....	II-19
2.9	Assembler Listings.....	II-20
2.9.1	Overview.....	II-20
2.9.2	Listing Controls.....	II-21
2.9.3	A Sample Listing.....	II-22
2.9.4	Multi-Module Listings.....	II-26
2.10	Downloading and Testing.....	II-26
2.10.1	Overview.....	II-26
2.10.2	Hardware Emulators.....	II-27
2.10.3	PROM Programmers and Target Boards..	II-27
3.	Creating Assembler Programs.....	II-28
3.1	Assembler Source Files.....	II-28
3.2	An Example Program.....	II-30
3.2.1	The ADEMO1 Source File.....	II-31
3.2.2	Comments.....	II-32
3.2.3	Modules.....	II-32
3.2.4	Segments.....	II-33
3.2.5	Symbols.....	II-35
3.2.6	Instruction Lines.....	II-37
3.2.7	The END Directive.....	II-39
3.3	Multi-Module Files and Libraries.....	II-39
3.3.1	Overview.....	II-39
3.3.2	A Library Application.....	II-40
3.3.3	Creating Multi-Module Files.....	II-41
3.3.4	Example Multi-Module File.....	II-42
3.4	Additional Assembler Topics.....	II-44
3.4.1	More About Segments and the 68HC11	II-44
3.4.2	Include Files.....	II-45
4.	Assembler Expression Reference.....	II-45
4.1	Constants.....	II-46
4.1.1	Integer Constants.....	II-46
4.1.2	ASCII Character Constants.....	II-47
4.1.3	Real Number Constants.....	II-47
4.3	User-defined Symbols.....	II-48
4.4	Expressions.....	II-49
4.4.1	Expression Basics.....	II-49
4.4.2	Delimiters in Expressions.....	II-51
4.4.3	TRUE and FALSE in Expressions.....	II-52
4.4.4	Relocatable Expressions.....	II-52
4.4.5	External expressions.....	II-53
4.4.6	Operator Precedence.....	II-53
4.4.7	Summary Table of Operators.....	II-54

4.5	Operator Reference	II-55
5.	Assembler Directive Reference	II-65
5.1	Summary of Assembler Directives	II-65
5.2	Module Directives	II-67
5.2.1	Overview of Module Directives	II-67
5.2.2	NAME	II-68
5.2.3	MODULE	II-68
5.2.4	ENDMOD	II-69
5.2.5	END	II-69
5.3	Symbol Directives	II-70
5.3.1	Overview of Symbol Directives	II-70
5.3.2	PUBLIC	II-71
5.3.3	EXTERN (or EXTRN)	II-71
5.3.4	LOCSYM	II-75
5.4	Segment Directives	II-75
5.4.1	Overview	II-75
5.4.2	ASEG	II-77
5.4.3	RSEG	II-78
5.4.4	COMMON	II-79
5.4.5	STACK	II-80
5.4.6	ORG	II-81
5.5	Value Directives	II-82
5.5.1	Overview	II-82
5.5.2	EQU (or =)	II-83
5.5.3	SET	II-83
5.5.4	DEFINE	II-84
5.6	Data Directives	II-85
5.6.1	Overview	II-85
5.6.2	RMB	II-86
5.6.3	FCB	II-86
5.6.4	FDB	II-87
5.6.5	FQB	II-88
5.6.6	FCC	II-90
5.6.7	ZPAGE	II-90
5.7	Addressing Modes	II-91
5.7	Conditional Assembly Directives	II-93
5.7.1	Overview of Conditional Assembly	II-93
5.7.2	IF	II-94
5.7.3	ENDIF	II-95
5.7.4	ELSE	II-95
5.8	Source Code Expansion Directives	II-96
5.8.1	Overview	II-97

5.8.2 \$ (Include file)	II-97
5.9 Listing Directives	II-98
5.9.1 Overview	II-98
5.9.2 LSTOUT	II-99
5.9.3 LSTCND	II-100
5.9.4 LSTCOD	II-101
5.9.5 LSTMAC	II-101
5.9.6 LSTEXP	II-102
5.9.7 LSTWID	II-103
5.9.8 LSTFOR	II-105
5.9.9 LSTPAG+, LSTPAG-	II-106
5.9.10 PAGESIZ	II-106
5.9.11 PAGE	II-106
5.9.12 TITL	II-107
5.9.13 PTITL	II-107
5.9.14 STITL	II-107
5.9.15 PSTITL	II-108
5.9.16 LSTXRF	II-108
6. Assembler Macros	II-111
6.1 Macro Basics	II-112
6.1.1 Macro Features	II-112
6.1.2 Macro Operation	II-112
6.1.3 A Simple Macro	II-113
6.1.4 Macro Parameters	II-115
6.1.5 Macro Guidelines	II-116
6.2 Advanced Macro Features	II-117
6.2.1 The Macro Expression Stack	II-117
6.2.2 Special Macro Symbols	II-118
6.2.3 Summary of Macro Operators	II-119
6.2.4 Macro Operator Reference	II-120
6.2.5 Unique Labels with a Macro	II-126
6.3 Macro Examples	II-127
III. Linker	III-1
1. An XLINK Overview	III-1
2. XLINK Characteristics and Features	III-2
3. Using the XLINK Linker	III-4
3.1 XLINK Installation	III-4
3.2 File Naming Conventions	III-5
3.3 Basic XLINK Operation	III-6

3.3.1	Command line Syntax	III-6
3.3.2	Notes for DEC, Sun and HP Users.....	III-10
3.3.3	Essential XLINK Options.....	III-11
3.3.4	Setting the Environment	III-15
3.3.5	Command line Examples.....	III-16
3.4	XLINK Command Files	III-19
3.4.1	Overview.....	III-19
3.4.2	XCL File Guidelines.....	III-20
3.4.3	Example XCL Files.....	III-21
3.5	More About How XLINK Works.....	III-24
3.5.1	Input Files and Modules	III-25
3.5.2	Libraries.....	III-26
3.5.3	Segment Location.....	III-27
3.5.4	Symbols and Typechecking	III-30
3.6	Linker Errors and Host Memory Management	III-31
3.6.1	Errors and Warnings.....	III-32
3.6.2	Host Memory Management.....	III-33
3.7	XLINK Listings.....	III-34
4.	Linker Command Reference"	III-39
4.1	XLINK Syntax Reference	III-40
4.1.1	General Syntax.....	III-40
4.1.2	Delimiters.....	III-40
4.1.3	Defaults.....	III-41
4.1.4	Upper/Lower Case	III-41
4.1.5	Command Files.....	III-42
4.1.6	Numeric Formats	III-42
4.1.7	Errors and Warnings.....	III-42
4.1.8	Notes for DEC, Sun and HP Users.....	III-42
4.2	XLINK Command Summary	III-43
4.2.1	XLINK Environment Variables	III-45
4.2.2	DOS Error Return Code.....	III-47
4.3	XLINK Command Detail.....	III-48
5.	XLINK Output Formats	III-76
5.1	Summary of XLINK Output Formats.....	III-77
5.2	Emulator/Development Tool Support	III-79
5.3	Output Format Details	III-79

IV. LIBRARIAN	IV-1
1. Introduction to the XLIB Librarian	IV-11
1.1 Library Basics.....	IV-1
1.1.1 Library Files	IV-1
1.1.2 C-6811 Programs and Libraries.....	IV-2
1.1.3 Assembler Programs and Libraries	IV-3
1.2. XLIB Librarian Features	IV-4
1.3 XLIB in the Development Cycle.....	IV-5
2. Using the XLIB Librarian	IV-6
2.1 XLIB Installation.....	IV-7
2.2 XLIB File Naming Conventions.....	IV-7
2.3 Basic XLIB Operation.....	IV-8
2.3.1 Running XLIB.....	IV-8
2.3.2 The XLIB Interactive Mode.....	IV-8
2.3.3 Common Librarian Tasks.....	IV-9
2.4 Librarian Batch Mode.....	IV-15
2.5 Librarian List Files	IV-17
3. XLIB Command Reference.....	IV-18
3.1 XLIB Command Summary.....	IV-19
3.2 XLIB Environment Variables	IV-20
3.3 XLIB Command Syntax.....	IV-21
3.3.1 Command Delimiters	IV-21
3.3.2 Command Abbreviations	IV-21
3.3.3 Command Defaults	IV-21
3.3.4 Specifying Modules.....	IV-22
3.4 XLIB Command Reference.....	IV-23
 Appendix A	A-1
A.1 System Requirements	A-1
A.1.1 Memory	A-1
A.1.2 Operating System.....	A-2
A.1.3 Disk Requirements.....	A-2
A.2 Software Installation	A-2
A.2.1 Modify Your CONFIG.SYS File	A-2
A.2.2 Copying the Disks to Your System.....	A-3
A.3.1 Installing the DOS files	A-3
A.3.2 DOS Error Return Values	A-5

Appendix B

Compiler Error and Warning Messages	B-1
B.1 C Error Messages	B-3
B.2 C Warning Messages	B-16

Appendix C

Assembler Error Messages	C-1
--------------------------------	-----

APPENDIX D

Linker Error and Warning Messages	D-1
D.1 Linker Error Messages	D-2
D.2 Linker Warning Messages	D-9

APPENDIX E

Librarian Error Messages	E-1
--------------------------------	-----

APPENDIX F

Libraries	F-1
-----------------	-----

Appendix G

Proliferation Chip Support	G-1
----------------------------------	-----

Appendix H

Special Emulator Support	H-1
--------------------------------	-----

Appendix I

Code Example	I-1
I.1 EXAMPLE.BAT	I-1
I.2 EXAMPLE.C	I-1
I.3 ASMFUNC.S07	I-3
I.4 EXAMPLE.XCL	I-4
I.5 CSTARTUP.S07	I-5

Appendix J

Glossary of Terms.....	J-1
------------------------	-----

Preface

Welcome to Archimedes Microcontroller C!

This manual describes how to use the Archimedes C-6811 Cross Compiler Software Development Kit for Motorola 68HC11 microcontrollers.

Product Overview

The Archimedes C-6811 kit is for software development of any 68HC11 proliferation chip. The kit includes a C-compiler, assembler, linker, librarian as well as a number of libraries. The object code produced by the Linker can be downloaded to a PROM-programmer, emulator, simulator, symbolic debugger or the target system.

The C-compiler is based on the traditional Kernighan and Ritchie C-standard. More importantly, it is also an implementation of the ANSI-standard for C-compilers. The C libraries include functions such as I/O and string support as well as advanced IEEE 32-bit/64-bit floating point support.

The assembler allows flexibility in speeding up time-critical sections of code as well as accessing certain hardware features. The linker links modules of code written in C and/or assembler and relocates code into absolute memory addresses.

The librarian is used to create and manage libraries. The linker's many output formats support most emulators and PROM-programmers.

Manual Overview

This manual is not a primer. It assumes a basic knowledge of C and assembly language programming. The manual is written to complement other literature, such as Motorola literature, which you already may have.

Start out by reviewing the tutorial and the release pages (colored pages at the back of the manual). If you are familiar with C and assembly programming as well as the 68HC11 architecture you may

start programming after you have also read through the C-chapter. If you are a true beginner, you should review the complete manual carefully before you get started. The manual is organized as follows:

Chapters

T	Tutorial
I	C
II	Assembler
III	Linker
IV	Librarian

Appendices

- A. Installation
- B. Compiler Error Messages
- C. Assembler Error Messages
- D. Linker Error Messages
- E. Librarian Error Messages
- F. Library Files
- G. Proliferation Chip Support
- H. Special Emulator Support
- I. Code Example
- J. Glossary of Terms

Please review the 'Tutorial' for other recommended literature. For your convenience, the software is not copy-protected. We trust your integrity in not making any illegal copies and only using the software on one machine at a time.

Again, welcome to Archimedes Microcontroller C!

The Time Saver

Tutorial



TM

ARCHIMEDES
SOFTWARE

TUTORIAL

T.1 What this Tutorial Covers

This software development kit supports any microcontroller chip in the Motorola 68HC11 family. For the purposes of this Tutorial, we will use "6811".

This section of the Archimedes C-6811 manual is designed to acquaint the first-time user with the basic principles of operation for getting up and running in a minimum amount of time. We will present an overview of C-6811 and walk through an example program to illustrate the use of some of the more important features. We will explain how to compile C-6811 code, link it with assembly language functions and create files for downloading to an emulator or PROM programmer. Other topics include modifying the CSTARTUP and putchar (character output) modules, interrupt handlers, debugging techniques, emulator support and common problems and their solutions.

The compiler (C-6811), linker (XLINK), assembler (A6801), and librarian (XLIB) that make up the C-6811 kit each has its own chapter that follow this Tutorial. As these chapters contain more detailed descriptions of the C-6811 use, the Tutorial does not attempt to cover every subject in full. We suggest that you review these other chapters after you have read the material presented here.

If you have used an Archimedes C compiler for another microcontroller (8051, 6301, Z80, 8096 etc.), this section will serve as an introduction to details that are specific to the 6811 family.

Because a thorough discussion of the 6811 MCU (microcontroller unit) and the C language is beyond the scope of this manual, we assume that the reader is familiar with the following subjects:

- The architecture and features of the 6811 family of microcontrollers. It is also necessary that you have at least a basic knowledge of the 6811 assembly language.
- The "C" language itself, as defined by Kernighan and Ritchie, or by the ANSI X3J11 "C" standard (see the references listed below).

We recommend the following books as references and learning aids for these subjects:

The Motorola Single-Chip Microcomputer Databook: Describes hardware features of the 68HC11 family.

Motorola Inc.
Microprocessor Division
Austin, TX 78721
512-928-6800

The C Programming Language, by Kernighan and Ritchie: The de-facto standard (and for years, the only) reference book on the "C" language. Also known as the "white book".

Prentice-Hall, Inc.
Englewood Cliffs
New Jersey, 07362

The "C" language series from Que Corporation are all highly recommended. Beginners will especially benefit from the first two titles:

The C Programming Guide, 2nd Edition
The C Self-Study Guide
Advanced C: Techniques and Applications
Common C Functions
C Programmers Library
Debugging C

Que Corporation
P.O. Box 50507
Indianapolis, IN 66250
Telephone orders: 1-800-428-5331

T.2 Standard Files and File Types

The C-6811 kit for an IBM PC/AT or compatible host consists of the two distribution diskettes and the documentation set. The actual files included with your kit will vary with the version number.

The general file types that are used and created with C-6811 are listed below. Please note that not all occurrences of these file types can be found on the distribution media (e.g., no .LST files are included):

*.EXE	Executable files (e.g., C-6811.EXE, the compiler)
*.C	C language source files
*.H	C header files, normally #included in C sources
*.INC	Assembly language include file (source)
*.R07	Libraries and other relocatable object files
*.S07	6811 assembly language source files
*.XCL	Extended command files for compiler and linker
*.LST	Default list file type used by compiler or linker
*.A07	Default output (absolute) file type used by linker
*.BAT	MS-DOS batch files
*.DOC	Extra documentation text files (may be included)

T.3 Installation

This release is equipped with an INSTALL utility which allows you to copy files from the release diskettes onto your hard disk. At the DOS prompt type `a:install <return>` and follow the instructions on the screen. See Appendix A for more information.

T.4 Other Development Tools

You may use any ASCII text editor, such as Brief, VEDIT, EMACS or WordStar, to create and modify C-6811 source and control files. Please note that when using a general-purpose word processing program, such as WordStar, to edit source code, be sure to set the editor in its "non-document" or plain ASCII mode of operation.

The XLINK linker directly supports a large number of output formats so that C-6811 absolute object code can be downloaded to almost any standalone development board, hardware emulator or PROM programmer for testing and creation of EPROMs. 6811 object code can also be tested on your host system using a 6811 software simulator. Full symbolic debugging capability is available with most emulators and simulators. See Appendix H for a list of development tools that are supported by C-6811. If the tool you are using is not shown, please call Archimedes Technical Support -- it may be compatible with one of those listed or support for it may have been recently added.

You may also want to make use of a native ANSI C compiler and debugger for your PC/AT host when developing code for C-6811. These include Microsoft C with CodeView, Borland Turbo C, Aztec C, and others. One of the advantages of ANSI-compatible C code is

that it may be easily ported from one target to another with minimal changes as long as standard rules of portability are observed (use standard I/O functions as much as possible, modularize programs, etc.).

The use of these native compilers allow the "generic" (hardware independent) portions of a C-6811 program to be compiled and tested entirely on the host system using a native compiler and debugger (e.g., Microsoft C and CodeView). Basic program logic, math functions, user interface routines, data structures, etc., can be verified this way. When the generic code is operating properly on the host system, it can be combined with 6811-specific portions and then compiled and linked with C-6811. Testing of the final program on the target system with an emulator or other debugging tool is thus minimized (this is often the most time-consuming part of development).

T.5 Archimedes C and the 6811

The C-6811 language is a complete implementation of the proposed ANSI C standard, which is actually a refinement of the K&R (Kernighan and Ritchie) definition. A set of the most common library functions is included to facilitate portability to other C compilers. In addition to these "standard" C features, Archimedes has added a number of features to support the development of 6811 microcontroller-based applications. The list below summarizes these additions; more details on each one can be found in the example code to follow and in the C-Compiler Chapter.

Multiple Memory Models: C-6811 supports three basic "memory models" to best match the hardware configuration of your target system:

small :	internal 6811/6801 RAM only (as in 6811 Mode 7)
large :	internal and external RAM (as in 6811 Modes 0-6)
banked:	internal and external RAM, banked-switched ROM

The banked model allows the creation of 6811 programs that exceed 64k in code size. See the C-Compiler chapter for details on how this feature can be implemented.

Note that when the small memory model is used, the compiler automatically uses the direct/short (8-bit) addressing mode when accessing static C variables, as they will reside entirely in the 256 bytes of internal RAM. Where external RAM is needed, the large or

banked model must be used, in which case the compiler uses extended (16-bit) addressing for all memory accesses.

The memory models are further divided into "static" and "reentrant" versions. In the reentrant models, memory for all local "auto" variables is dynamically allocated on the stack when a function is called and freed when the function exits. In the static models, auto variables are forced by the compiler to reside in a static memory area.

The reentrant models must be used when an interrupt handler is written in C or when a function is called recursively (i.e., it calls itself). The static models are used when execution time is the critical factor as access to the stack is minimized.

The complete set of six C-6811 memory models are listed below with their associated compiler switches. See the C-Compiler Chapter for more information on selecting the proper one for your application:

-ml	large/reentrant (default)
-ml -d	large/static
-ms	small/reentrant
-ms -d	small/static
-mb	banked/reentrant
-mb -d	banked/static

PROMable Code: The compiler can generate PROMable code in which the initializers for static variables will be stored in ROM and then copied over into their RAM variables at run time. The -P compiler option is used to enable PROMable code.

Direct Hardware Access: C-6811 allows direct access from C to on and off-chip hardware features without the need to call assembly language functions. Reading and writing an on-chip register or I/O port is accomplished through the use of C pointers:

```
c = *(char *)0x12;    /* read the byte at location 12 */
*(char *)0x8000 = c;  /* write a byte to external location 8000h */
```

Note

*The construct (char *) is a "cast", and is needed when an integer constant or variable is used as a pointer. C-6811 requires a cast when assigning integers to pointers, and vice-versa.*

The 6811 ports and registers are pre-defined in a file called IO6811.H, so they can be accessed symbolically:

```
/* in file IO6811.H */
#define RDR (* (char *) 0x12) /* receive data register */
.
/* usage in C program */
#include <io6811.h>
char c;
.
c = RDR; /* read the serial I/O receiver */
```

C-6811 also includes three "in-line" C functions to control the interrupt system of 6811:

enable_interrupt

```
void enable_interrupt(void); /* generates a CLI instruction */
```

disable_interrupt

```
void disable_interrupt(void); /* generates an SEI instruction */
```

wait_for_interrupt

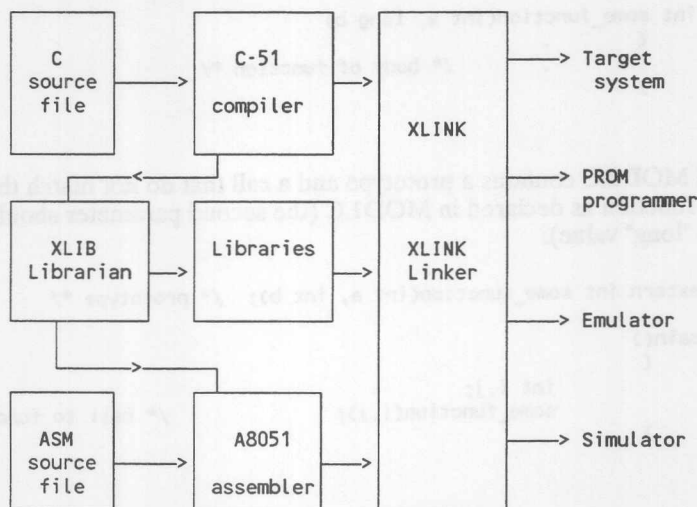
```
void wait_for_interrupt(void); /* generates WAIT instruction */
```

Please note that these functions act like C-6811 keywords, or extensions to the C language and should not be declared as normal functions (i.e., a function prototype is not required). They are not called as functions but generate in-line code.

For compatibility, the `-e` compiler option must be used to enable the recognition of these inline functions.

T.6 The Development System

The C-6811 kit consists of a compiler, assembler, linker and librarian programs. The diagram below illustrates how these are used together to develop an 6811 application. An explanation of each program follows:



Compiler

The C-6811 compiler is a single program that reads C source (filetype `.C`) and header (`.H`) modules as input and directly produces a relocatable object file (`.R07`) as output, which can then be processed by the linker. Because of this single-pass design, a separate assembly phase is not required. The output file is in an Archimedes-proprietary format called (Universal Binary Relocatable Object Format) that contains code, data and symbolic information. By using compiler switches, various listing files can be produced as well, including one with intermixed C and assembler statements (`-L-q` options, `.LST` filetype). A straight assembler source file (`.ASM`) can also be generated (`-a`) that allows the programmer to modify and re-assemble the compiler output to further optimize the code, if required.

In addition to the standard typechecking required by ANSI C, the C-6811 compiler includes a very useful built-in program verifier (-g option) similar to the Unix "LINT" utility that checks your source code for potentially erroneous (though legal) conditions such as unreferenced local variables, unreachable code or constant array indexing that is out of range.

When enabled, this option also causes typechecking information to be included in the output file for all external references so that the linker can verify that the interface between modules is consistent. For example, say source file MOD1.C contains a function declaration:

```
int some_function(int a, long b)
{
    .
    .
    .
    /* body of function */
}
```

And MOD2.C contains a prototype and a call that do not match the real function as declared in MOD1.C (the second parameter should be a "long" value):

```
extern int some_function(int a, int b); /* prototype */

main()
{
    int i,j;
    some_function(i,j);
    /* call to function */
}
```

When these two modules are linked together, XLINK would print an error message (assuming both modules were compiled with the -g switch):

```
Warning[6]: Type conflict for external/entry some_function,
in module mod2 against external/entry in module mod1
```

Assembler

The 68HC11 assembler uses the A6801 macro assembler and reads a standard 6811 assembler source file as input (.S07 default filetype) and creates a relocatable object file in the UBROF format as output (.R07). A list file with cross-references may be also generated (.LST). It operates as a single, two-pass program. A6801 is highly compatible with assemblers from other vendors.

The assembler is provided so that time-critical sections of code may be written in 6811 assembly language to minimize the amount of software overhead. For example, a timer interrupt handler that is invoked every few hundred microseconds should be coded in assembler. If desired, the C-6811 compiler output (with the -a switch) can be reassembled after further optimization, though this is purely optional.

Linker

XLINK is a flexible linker/loader that reads as input relocatable UBROF-format object files (.R07 filetype) created by the compiler or assembler. The -Z linker command is used to set the base addresses for the various code and data segments to match the ROM and RAM memory maps of your target system. As output, XLINK produces an absolute object file with optional symbols or symbol file in one of a number of different formats to support most emulators, simulators and PROM programmers, including: Motorola S-records, Intel HEX, Microtek, Tektronix, HP and more than 30 others.

A listing file (.MAP) can also be produced. This .MAP file contains a module map, a segment map, and a cross-reference listing of global symbols. This information can be used for debugging to locate functions and global variables when a symbolic emulator/debugger is not available.

Two types of modules can be loaded with XLINK. One type is a "program" module which will always be loaded (i.e., placed in the output file image) when the name of the file in which it is located is processed by XLINK. The CSTARTUP module (which contains essential run-time initialization code) and most user-written C modules will have the program attribute.

The other type is a "library" module, which is loaded only when it contains an entry (a function or global variable name) that has been referenced by another module. This has the effect of only loading modules from a library file that are really needed, thus minimizing memory usage and reducing link time. The standard C-6811 library files CL6811.R07 and CL6811B.R07 (for large/medium, small, and banked models, respectively) all contain modules that have the library attribute. CSTARTUP is the only module in the library file that has the program attribute, so it is normally the only one that is always loaded when the library file is processed.

XLINK does not allow linking C-6811 or A6801 object modules with those produced by compilers or assemblers from other vendors. However, most assembly programs can be easily converted and reassembled with the Archimedes A6801.

Librarian

The XLIB utility is used to manage the standard C-6811 library files and to create user libraries. XLIB operates on relocatable object files (filetype .R07) and permits listing, renaming, replacing and deleting modules, segments and entries. A module may also be given a "program" or "library" attribute with XLIB.

T.7 Putting Together the Example Program

We will now demonstrate the use of the C-6811 kit by walking through the example program provided in source form on the distribution diskettes. This program illustrates the concepts and procedures that will help you create your own C-6811 programs.

In creating a C-6811 program for the first time, these steps should be followed as a general rule:

1. Examine, and modify if needed, the CSTARTUP module. Replace the original version with the modified one in the library file if any changes are made (our example program does not require any CSTARTUP changes).
2. If the standard character output functions are needed (printf and/or putchar), you must modify the PUTCHAR.S07 assembler routine to support your hardware configuration. Replace the putchar supplied in the library (which is for the Motorola BUFFALO monitor program) with the modified version. Do the same for character input (getchar) if this function is needed.
3. Edit and compile your C source file(s). If desired, you can compile and test the "generic" (non-6811 specific) portions of the program with a host-resident compiler (e.g., Microsoft C with Codeview).
4. Edit and assemble any assembly language source file(s).

5. Modify one of the provided linker command files (LNK6811?.XCL) to match your ROM/RAM memory map, select the program modules to be loaded and set the output file format to that used by your emulator, simulator or PROM programmer.
6. Download the absolute object file to your hardware emulator, simulator, target board or PROM programmer for testing.

We will now build the example program using these steps as a guideline. The complete source files are included on the distribution disks and listed in Appendix I; only partial listings are shown below.

T.7.1 Modifying and Assembling CSTARTUP

A number of critical C-6811 system functions are located in the CSTARTUP assembler module:

- Stack pointer initialization code
- DATA segments initialization inline code
- User-added 6811 RESET vector, interrupt vectors, and interrupt handlers
- User-added hardware initialization code
- Call to the main() program function
- Jump to the Exit routine

A standard version of CSTARTUP is provided in the library files. The source code for this routine is also provided. Typically, this module is modified by the user only when application specific code is needed. If CSTARTUP is modified, it should be assembled and put back in the C standard library, replacing the original one. In any case, it is helpful to understand what CSTARTUP does and why.

We will now examine portions of the CSTARTUP.S07 file, which is the assembly language source for the version of CSTARTUP provided in the standard C libraries (CL6811.R07 etc.). Since our example program is designed for the large model, those portions of CSTARTUP that apply to the small or banked models will not be shown. Refer to the disk file for a complete listing:

NAME	CSTARTUP	
\$defcpu.inc		
EXTERN	?C_EXIT	Where to go when program is done
EXTERN	main	Where to begin execution

The NAME statement not only names the module but also gives CSTARTUP the program attribute, which means it will always be loaded when the file in which it is located (the library file) is processed by the linker. The MODULE statement is used instead of NAME when a LIBRARY (conditional load) is desired.

The "\$DEFCPU.INC" statement tells the assembler to include the definition file for the proper CPU and enables banking. This file contains various DEFINES that control the conditional assembly of parts of the module. As part of the configuration process, one of the following files should be copied to the file DEFCPU.INC:

```
DEF6811.INC      6811 CPU type
DEF6811B.INC     6811 with bank-switched ROM
```

For example, for a 6811 non-banked system enter the following command before assembling a modified CSTARTUP file:

```
C>COPY DEF6811.INC DEFCPU.INC
```

The next section of CSTARTUP code defines the stack holding segment CSTACK without allocating any storage space for it. The stack size is defined in the linker command file (.XCL). The stack is used for return addresses, temporary storage, dynamic storage of local/auto variables for C functions (reentrant models only) and to pass parameters to functions.

```

RSEG      CSTACK      Use data segment CSTACK
RMB       0           A bare minimum
stack_begin:

RSEG      RCODE       Use CODE segment RCODE
init_C:
LDS       #.SFE.(CSTACK)-1    From high to low addresses
```

The RCODE segment is normally the first code segment to be linked. After a CPU reset, program execution begins at "init_C" with the initialization of the stack pointer.

```
BSR      seg_initThe old ?SEG_INIT_L07
```

seg_init is a subroutine that initializes the data segments. Static initializers, located in ROM, are copied into their corresponding variables or strings, located in RAM. All uninitialized variables are also set to zero, according to the ANSI standard. If you do not need statically initialized variables or strings, this subroutine (or the its call) may be deleted.

It is sometimes necessary to perform hardware initialization in assembly code or enable interrupts before entering the main C program. This is normally the place to insert such code. Note, however, that if your hardware design includes bank-switched RAM, you must enable the proper bank BEFORE the call to ?SEG INIT L07. (C-6811 does not directly support bank-switched RAM, but this can be done explicitly by the user).

* Insert hardware initialization code here...

```
JSR    main          main() C program entry point
JMP    ?C_EXIT
```

One place where hardware must be initialized from CSTARTUP is the case of the 68HC11 OPTION register, which must be set up within 64 E-clock cycles after a CPU reset. For example, this code could be added to CSTARTUP to enable the COP and clock monitor features:

```
init_C:                                Execution starts here
    LDAA    #$9B                        Enable COP, clock monitor, etc.
    STAA    $1039                       Load OPTION register
    LDS     #.SFE.(CSTACK)-1
    .
    .
```

Finally, a JSR is made to the C program entry point, the main() function. When the main function terminates, execution jumps to the Exit routines which is simply an infinite loop. If the BUFFALO monitor is used, an SWI instruction causes a software interrupt which returns control to the debugger/monitor. Since most microcontroller applications run continuously, an error routine could be placed here to trap an unexpected exit.

The following few lines of CSTARTUP can be used to create a vector for processor RESET at location \$FFFE. Since we want the "init_C" routine to be executed first after a reset, its address is placed at this location.

```

COMMON  INTVEC
IF      proc6811
*
RMB     40                                Assuming start address = FFD6 for 68HC11
ELSE
*
RMB     20                                Assuming start address = FFEA for 6301
ENDIF
FDB     init_C
ENDMOD  init_C      'init_C' is program entry address

```

If an interrupt vector needs to be defined without the interrupt C keyword, it can be added before the RESET vector. The handler can be written in assembler or C. For example, if a timer overflow interrupt is to be written in C without using the interrupt C keyword, the following code can be used:

```

EXTERN  timer_int      External C routine
COMMON  INTVEC
*
RMB     28              Assuming start address = FFD6 for 68HC11
FDB     handler         $FFD6+28=$FFF2
RMB     10              $FFD6+28+2
FDB     init_C          $FFD6+28+2+10=$FFFE
RSEG RCODE
handler:
JSR     timer_int      Call C handler
RTI
ENDMOD  init_C          'init_C' is program entry address

```

Refer to the appropriate microcontroller handbook for a list of interrupt vector locations. Note that there is no need to save the working registers (A, B, ...) as this is done by the processor at the time of an interrupt acknowledge.

If changes or additions are made to the CSTARTUP.S07 file, it must be reassembled and put back in the library (replacing the original version). Since our example program is for a non-banked 6811 application, the CL6811.R07 library file is used:

```
C>A6801 CSTARTUP

C>XLIB
  *DEF-CPU 68HC11                (use 68HC11 CPU)
  *REP-MOD CSTARTUP CL6811
  *EXIT
```

CSTARTUP does not have to be loaded from the library -- it can be linked in explicitly as a standalone module if desired. However, be sure to delete the original CSTARTUP module from the library first to prevent it from being automatically loaded as well (see the Librarian Chapter).

T.7.2 The PUTCHAR and GETCHAR Routines

Like CSTARTUP, the putchar (character output) and getchar (character input) functions must usually be custom tailored for your target hardware if you intend to use standard character I/O. Since the printf (formatted print) function calls putchar indirectly, it will work when putchar works. If your application does not call for standard character I/O, there is no need to modify these functions and you may skip this section.

The versions of putchar and getchar that come installed in the standard libraries are written to work with the Motorola BUFFALO monitor/debugger. The source code for these functions can be found in the PUTCHAR.S07 and GETCHAR.C files. PUTCHAR.S07 is listed below for reference:

```

*-----*
*                                     *
*      PUTCHAR.S07                   *
*                                     *
*      int putchar(int value);       *
*                                     *
* This module contains an example of a "putchar" *
* routine that is user-modifiable.    *
*                                     *
* Shown here is an empty "shell" not adapted for *
* any particular hardware interface.  *
*                                     *
* Version 3.00 [A.R. 10/Aug/88      *
*-----*
MODULE    putchar
$defcpu.inc
PUBLIC    putchar($16,$80)

IF        banking

EXTERN    ?X_RET_L09
RSEG      FLIST
putchar:
    FQB    bank_putchar
    RSEG    CODE
bank_putchar:
    TSX
    LDAA    4,X                Put character in the A-register
    *      Place to put hardware dependent code
    *      that prints the character in A

    JMP     ?X_RET_L09

ELSE

RSEG      CODE
putchar:
    TBA                Put character in the A-register
    *      Place to put hardware dependent code
    *      that prints the character in A

    RTS

ENDIF

END

```


After making your changes, the PUTCHAR.S07 module should be assembled and put in the library replacing the distribution version:

```
C>A6801 PUTCHAR (source_file)
C>XLIB
*DEF-CPU 68HC11
*REP-MOD PUTCHAR CL6811
*EXIT
```

If you explicitly link in your modified putchar module (instead of putting it in the library), be sure to delete the old version from the library first (use the `XLIB DELETE-MODULES` command).

We will now look at our `EXAMPLE.C` program which is supplied on the distribution diskette and listed in its entirety in Appendix I. Most C-6811 programs will follow this same general structure, which has the form:

```
#include header files
#define constants and C-macros
declare global and external variables
prototypes for external functions

function1
    declare local variables

function2
    declare local variables
.
.
main program body
    declare local variables
```

The first section of the example program is shown below:

```
#include <stdio.h>          /* header for printf and putchar */
#include <iolib6811.h>      /* header with internal address definitions */

#define TRUE 1
#define FALSE 0

extern int asmfunc (int parm1); /* declaration for sample assembly routine -- see ASMFUNC.S07 */
```

The IO6811.H file contains #defines for the 6811 I/O ports and registers. This header file should be included whenever access to these ports is required.

The last line is an ANSI prototype for an external assembly language function contained in the file ASMFUNC.S07, which will be assembled and linked with our main program. This prototype defines the exact number and type of formal parameters and the return value so that the compiler and linker can check the interface between modules. All functions located outside the calling source module (i.e., in another file) should be declared in this manner.

Likewise, a global variable that is to be referenced in more than one source file should be declared as "extern" in all modules except the one where it is defined and its storage space allocated. For example, if a function contained in another source file need to have access to the global variable "count", that file should contain the following external declaration:

```
extern int count; /* now accessible to this entire source file */

int function1(int i)
{
    if(i = count)
        return(1);
    .
    .
}
```

If there is a large number of global variables, their 'extern' declarations can be placed in a header (.H) file which is then #included in the source modules that need it.

The use of C pointers allows easy access to absolute external memory locations, as would be needed for an external memory-mapped I/O device such as a UART. However, ANSI C does not permit the direct use of integer constants as pointers, or the assignment of integers to pointers. To do this, a cast of the integer constant (e.g., the UART's address) must be performed first, as shown in the #define statement below.

The function `dump` prints the contents of the 6811 internal RAM, the first byte of EEPROM, and the value of two I/O ports:

```
#define EXT_PORT (*(char *) 0xA000) /* defines port at A000 hex */
#define EEPROM (char *)0xB600    /* ptr to EEPROM base */
void dump() /* display RAM, I/O ports, etc. */
{
    int line, byte;
    /* dump internal RAM */
    printf("6811 internal RAM dump:\n");
    for(byte = 0; byte < 16; printf("%X ", byte++));
    for(line = 0; line < 16; line++)
    {
        printf("\n%02X: ", line * 16);
        for(byte = 0; byte < 16; byte++)
            printf("%02X ", *(char *)((line * 16) + byte));
    }
    /* read PORT A, PORT E and an external port */
    printf("\nPORT A = %02X \nPORT E = %02X\n", PORTA, PORTE);
    printf("Port or memory at address A000h = %02Xh\n", EXT_PORT);
    printf("First byte in EEPROM = %02X\n", *EEPROM);
}
```

The routine `eprom_write` is used to perform the sequence required to write data to the EEPROM device (see the 68HC11 hardware manual for more information):

```
void delay(void) /* delay of about 12 ms needed */
{
    int i;
    for (i = 0; i < 400; i++);
}

void eprom_write(char data, char *address)
{
    PPROG = 0x16; /* single-byte erase mode */
    *address = 0xff; /* write anything */
    PPROG = 0x17;
    delay();
    PPROG = 0x16;
    PPROG = 0x02; /* program mode */
    *address = data; /* write data */
    PPROG = 0x03;
    delay();
    PPROG = 0x02;
    PPROG = 0; /* read mode */
}
```

The main program uses an inline function to disable interrupts, then enters a continuous loop where it writes a byte to the EEPROM, calls the dump routine and the assembler function, then runs the sieve benchmark (not listed here):

```
void main() /* main example.c program */
{
    int parm1;
    disable_interrupt();
    printf("EXAMPLE.C sample program for 68HC11.\n");
    while(1)
    {
        eeprom_write(0xA5,EEPROM); /* write A5h to first byte of
EEPROM */
        dump(); /* display memory, ports */
        parm1 = 10; /* set up parameter */
        asmfunc (parm1); /* call sample assembler routine */
        sieve(); /* do the Sieve */
    }
}
```

T.7.4 Compiling the Program

The example program is compiled with C-6811 as shown below. We suggest that you try this on your host system and examine the resulting map file (compile an unmodified copy of EXAMPLE.C for now):

```
C>C-6811 example -ml -e -g -L -q (source_file options...)
```

```
Archimedes 68HC11 C-Compiler V3.20B/DOS
(c) Copyright Archimedes 1991
```

```
Errors: none
Warnings: none
Code size: 468
Constant Size: 241
Static Variable Size: Data(2049) Zpage(0)
```

The first parameter in the command line following the C-6811 program name is the name of the C source file; a filetype of .C is assumed, so it is normally left off. The EXAMPLE.C file should be in the current DOS directory; a directory path may be used to specify an alternate directory for the source file (e.g., \ARCH\SOURCE\example).

The standard header files (STDIO.H, IO6811.H, etc.) should be in the current directory or in the directory defined by the C INCLUDE environment variable. Alternately, the search directory for #include files can be specified on the command line with the -I option. Please note that the search directory name must have a trailing backslash (\), as in:

```
C-6811 example -ml -e -g -P -L -q -I\c6811\include\
```

Two output files are created by this run of C-6811. One file is EXAMPLE.R07 which contains the relocatable object code (Archimedes proprietary UBROF format), ready to be linked. The second file is EXAMPLE.LST which is a C/assembler list file. As C-6811 is entirely memory-resident, no temporary files are created and compile speed is very high.

The "Code size" refers to the amount of inline code produced by the compiler. When the program is linked with the standard library, a number of runtime "helper" routines will be added, as will other C and assembler modules and user-called library functions (putchar, printf, etc.). The final code size is determined by examining the linker's segment or module map.

The compiler "switches" that follow the source file name are the most-commonly used compiler options, so we will discuss each one. A full list of compiler options appears in the C-Compiler Chapter, or simply type "C-6811" with no parameters to display the compiler help message. These options can appear before or after the source file name in the command line.

- ml Selects the large reentrant memory model (the default, so this option could have been left out). The large model supports up to 64k of external memory (ROM + RAM). Refer to the C-Compiler Chapter for a more complete discussion on selecting the proper memory model for your application.
- e Enables the recognition of C-6811 extensions to the ANSI C language. This option allows the following inline C functions to be used: enable_interrupt, disable_interrupt, wait_for_interrupt.

- g Enables the strict global typechecking feature which warns about the use of certain legal, though possibly dangerous, constructs in your code. We strongly suggest the use of this option as it helps eliminate many types of common C errors. It also enables the generation of typing information in the output file so the linker can check the interface between modules.
- P Generates PROMable code, where static initializers for variables are copied from PROM to the variables in RAM at run time by the ?SEG_INIT_L07 function (see CSTARTUP).
- L,-q These two options, normally used together, cause the generation of a list file named EXAMPLE.LST that contains a mixture of C code (-L) and the corresponding assembly-language statements (-q). This file is useful for debugging purposes as well as just studying the compiler's output. A small portion of EXAMPLE.LST is shown below. We suggest that you study the complete list file after compilation to get an idea on how C-6811 produces code, uses segments, and allocates data:

```

59          printf("%02X ",*(char *)((line * 16) + byte));
\ 00A3 30          TSX
\ 00A4 EC00        LDD      0,X
\ 00A6 05          ASLD
\ 00A7 05          ASLD
\ 00A8 05          ASLD
\ 00A9 05          ASLD
\ 00AA E302        ADDD     2,X
\ 00AC 8F          XGDX
\ 00AD E600        LDAB     0,X
\ 00AF 4F          CLRA
\ 00B0 37          PSHB
\ 00B1 36          PSHA
\ 00B2 CC002B      LDD      #?0019
\ 00B5 BD0000      JSR      printf
\ 00B8 38          PULX
\ 00B9 30          TSX
\ 00BA 6C03        INC      3,X
\ 00BC 2602        BNE      **+4
\ 00BE 6C02        INC      2,X

```

You will notice in the listing some labels that are called that begin with a "?". These are C-6811 "helper" routines that are needed to implement various runtime functions that cannot be produced efficiently with inline code. The compiler automatically produces these calls transparently to the programmer. Some of these functions can be useful in writing assembly language programs that will interface with C.

Another option, -a, produces a pure assembly language source file that can be reassembled if necessary, though there is usually little need for this.

T.7.5 An Assembly Function

While most microcontroller applications can be coded completely in C, it is often necessary to write small portions in assembly in order to increase the code efficiency (execution speed). This is most often true with timer, and other high-speed, interrupt handlers. ASMFUNC.S07 is the source for a "dummy" assembly module that demonstrates how to create a C-callable assembly function:

```
*-----*
*   asmfunc.s07 -- dummy assembler function   *
*                                           *
*   int asmfunc(int parm1);                 *
*                                           *
*   This is a dummy routine that demonstrates how *
*   to interface C with an assembly function.  *
*   Called from example.c program.           *
*                                           *
*-----*
MODULE    asmfunc
PUBLIC    asmfunc

P68H11          Use for 68HC11 chip
RSEG          CODE
asmfunc:
*   Insert your assembler routine here.
*   The first parameter (parm1 in this case) is now
*   in the A:B register pair (B is the low byte).
*   This first parameter, and any following parameters
*   that were declared, are also on the stack and may be
*   pulled off after saving the return address, which is
*   on the top of the stack. (see section 1.9 in the manual).

*   A return value, if needed, should be placed in the B or
*   A:B registers.

RTS          Return to C program
END
```


When control is passed to an assembly routine, the first formal parameter is in the B (8-bit char) or A:B (16-bit int, pointer) register. If there is more than one parameter, the second, and any further, parameters are located on the stack following the return address and must be pulled off. Char and int return values are placed in the B and A:B registers, respectively. There is no need to save any of the 6811 registers (A, B, IX, CCR) when calling an assembly function.

Refer to the C-Compiler Chapter for complete details on assembly language interface.

The ASMFUNC.S07 file is assembled as follows:

```
C>A6801 ASMFUNC ASMFUNC      (source_file list_file)
```

```
Archimedes 6801 Assembler V2.00/DOS
```

```
(c) Copyright Archimedes 1991
```

```
Errors:  None      #####
Bytes:   1         # asmfunc #
CRC:     8034      #####
```

Two files are created by the assembler: ASMFUNC.R07, the relocatable object file, and ASMFUNC.LST, a list file.

T.7.6 Linking the Example Program

XLINK is now used to link together the EXAMPLE.R07, ASMFUNC.R07 and CL6811.R07 (non-banked 6811 library) relocatable object modules into the final, absolute program. Try linking the example program under control of the EXAMPLE.XCL command file, as shown:

```
C>xlink -f example
```

```
Archimedes Universal Linker V4.38/DOS
```

```
(c) Copyright Archimedes 1991
```

```
Errors: none
Warnings: none
```

This link operation will produce an absolute object file, EXAMPLE.A07, and a cross-reference/map file, EXAMPLE.MAP. When linking C code, XLINK should always be used with the aid of a command file (filetype .XCL) that contains the input file names and

the switches. XLINK can accept all of its input file names and options on the command line, but it should only be used in this manner when linking small assembly language programs.

The EXAMPLE.XCL command file that comes with C-6811 is shown below:

```
-!                                     -LNK6811.XCL-

XLINK 4.xx command file to be used with the 68HC11 C-compiler V3.x
using the -ms or -ml options (non banked memory models).
Usage: xlink your_file(s) -f lnk6811

First define CPU  -!

-c68hc11

-! Allocate segments which should be loaded -!

-! First allocate the read only segments.
C000 was here supposed to be start of PROM -!

-Z(CODE)RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=C000

-! Then the writeable segments which must be mapped to a RAM area
2000 was here supposed to be start of RAM.
Note: Stack size is set to 128 (80H) bytes with 'CSTACK+80'
!

-Z(DATA)DATA,IDATA,UDATA,ECSTR,WCSTR,TEMP,CSTACK+80

-! The interrupt vectors are assumed to start at FFD6, see also
CSTARTUP.S07 and INT6811.H.

-Z(DATA)INTVEC=FFD6

-! NOTE: In case of a RAM-only system, the two segment lists may be
connected to allocate a contiguous memory space. I.e. :
-Z...CCSTR,DATA...=start_of_RAM -!

-! The segment SHORTAD is for direct addressing, let XLINK check
that the variables really are within the zero page -!

-Z(DATA)SHORTAD=00-FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write

-! See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read

-! Now load the 'C' library -!
cl6811 -! or cl6811d -!
```

```

example
asmfunc

-! Code will now reside in example.a07 in MOTOROLA 'S' format -!
-o example.a07
-! Create a map file with a full cross reference list -!
-x
-l example.map

```

Each of the most-often-used XLINK options is explained below. For a complete list and detailed description, refer to the Linker Chapter of the manual, or type "XLINK" with no parameters for a help message. The order of the switches in the file is of no importance as the linker scans them all first before processing the input files. Commands may also be entered on the XLINK command line from DOS.

- ! This is used to bracket comments, which may extend several lines. Be sure to leave a space between the '-' and the adjacent word in each comment.
- c Used to define the CPU type, which is 68HC11 in our case (which includes the 6811, 6801, 6301 and compatible CPUs).
- Z This command is used to define the type and base address of the various code and data segments used by C-6811. There are two lines using the -Z command in EXAMPLE.XCL:

```
-Z(CODE)RCODE, CODE, CDATA, ZVECT, CONST, CSTR, CCSTR=C000
```

The (CODE) designator at the beginning of the line, which is optional, tells the linker that these segments have a type of "CODE" so it can verify that the segment names that follow all have this same type. If you attempt to link a segment that has "DATA" type, an error message will be generated.

The segment names listed here (RCODE, CODE, etc.) are the predefined names that C-6811 uses for various types of executable code and constant (read only) data. If you examine the list-file output from the compiler using the -L and -q options, you can see how these segments are used. A list of these segments and their functions can be found in the C-Compiler Chapter.

The number at the end of the list is a hexadecimal address that determines where the first of these segments will be linked. In our case, we want RCODE to be located at C000h. In a ROM-based target, this address would be set to the start of ROM. The second segment (CODE) will start immediately following the end of the first, and so on, up to the last code segment.

```
-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK
```

This command determines where the data segments used by C-6811 will reside in memory (RAM). The (DATA) designator at the beginning of the line, which is optional, tells the linker that these segments have a type of "DATA". The segment names listed here (DATA, TEMP, IDATA, etc.) are the predefined names that C-6811 uses for various types of read/write storage, including variables, strings, and stack. If you examine the list-file output from the compiler using the -L and -q options, you can see how these are used. A list of these segments and their functions can be found in the C-Compiler Chapter. It is best to ALWAYS USE this standard list of segment names when linking C code, even if you believe that some segments will not be needed by your program.

In our example, the data segment list is not followed by an address, which means that these segments will be linked following the code segments (see the segment map). In a typical target system, the data segments would be based at the beginning of RAM.

example, asmfunc, cl6811

These are the names of the example files to load and link; a filetype of .R07 is assumed. A module with a "program" attribute will always be loaded from a file when it is processed by XLINK. Since C modules are "programs" by default, the module EXAMPLE will be force loaded. Since the ASMFUNC module contains a NAME statement, it, too, will be considered a "program" and will be loaded.

However, the standard C-6811 library file, CL6811, contains many modules, most of which have the "library" attribute, so only those modules that have entries (functions) that are referenced somewhere will be loaded. This includes user-invoked C functions (printf, putchar, etc.) and runtime helper routines (? functions). The one exception in CL6811 is the module CSTARTUP, which will always be loaded since it contains a NAME statement, which gives it the "program" attribute. Therefore, only those library functions needed for a given program will be included in the link process.

-l, -x

These options tell the linker to generate a complete cross-reference listing file for the program including an absolute memory map of where all the segments and modules are located. This is a useful tool for determining RAM and ROM usage and for debugging the final program. The -l option is used to specify the listing filename (EXAMPLE.MAP). If -l is omitted, the listing will be sent to the screen.

A section of the EXAMPLE.MAP file created by the linker is shown below; the module map is followed by the segment map:

```
FILE NAME : example.r07
PROGRAM MODULE, NAME : example

SEGMENTS IN THE MODULE
=====
CODE
Relative segment, address : C26D - C42C
ENTRIES      ADDRESS      REF BY MODULE
delay        C26D         Not referred to
dump         C2B9         Not referred to
eeprom_write C282         Not referred to
main         C407         CSTARTUP
sieve        C361         Not referred to
```

```

DATA
Relative segment, address : C530 - CD30
ENTRIES
flags                ADDRESS          REF BY MODULE
                     C530             Not referred to
LOCALS
                     ADDRESS
?0023                C530
?0024                CD31

```

SEGMENT	START ADDRESS	END ADDRESS	TYPE	ORG	P/N	ALIGN
=====	=====	=====	=====	=====	=====	=====
RCODE	C000	- C0AD	rel	stc	pos	0
CODE	COAE	- C42D	rel	flt	pos	0
CDATA	Not applicable		rel	flt	pos	0
ZVECT	C42E	- C431	rel	flt	pos	0
CONST	C432	- C442	rel	flt	pos	0
CSTR	C443	- C52F	rel	flt	pos	0
CCSTR	Not applicable		rel	flt	pos	0
DATA	C530	- CD30	rel	flt	pos	0
IDATA	Not applicable		rel	flt	pos	0
UDATA	Not applicable		rel	flt	pos	0
ECSTR	Not applicable		rel	flt	pos	0
WCSTR	Not applicable		rel	flt	pos	0
TEMP	Not applicable		rel	flt	pos	0
CSTACK	CD31	- CDB0	rel	flt	pos	0
INTVEC	FFD6	- FFFF	com	stc	pos	0
SHORTAD	Not applicable		dse	stc	pos	0

-o example.a07

The output file name is set with the -o option. Here, the absolute output file is named EXAMPLE.A07. Since we have not specified an alternate output format, the default format for the 6811 of the Motorola S-Records.

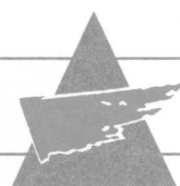
T.7.7 Downloading and Testing

The final output file, EXAMPLE.A07, can now be downloaded and tested on the Motorola M68HC11 EVB evaluation board or your hardware emulator, if desired. Refer to the documentation provided with your target board, emulator or PROM programmer for specific instructions on downloading files.

Note that if an alternate output file type is specified (-F option) that contains symbolic information, only global-type symbols will be included in your emulator's symbol table (function names, global static variables, PUBLICed assembler labels etc.). If you need to see local static (compiler-produced) symbols, the C-6811 "-r" option should be used when compiling a C program.

The Time Saver

C-Compiler



TM

ARCHIMEDES
SOFTWARE

C-6811 COMPILER

1. Overview

1.1 Introduction

The 'C' language, invented by Brian W. Kernighan and Dennis M. Ritchie (K&R), has become a major system-programming language for applications ranging from simple controller-type programs to complete operating systems. Its unique portability assures that code can be moved between radically different architectures with considerably less effort than is possible with other 'standard' languages such as Pascal and FORTRAN.

After its birth in the early seventies the, C language has evolved creating certain 'dialects' which have sometimes been incompatible with the original C definition (K&R). In order to regain control of the C language, the American National Standards Institute (ANSI) has set up a group of industry software experts to update the language definition. The Archimedes C development team is involved as an observer in this group and the compiler described in this document closely follows the standard.

Most of the ANSI additions are minor changes to enhance portability with one notable exception: function prototyping. Function prototyping allows C users the option of full parameter checking, as in Pascal, coupled with the C language assignment conversion rules. The Archimedes C is based on the K&R-standard with ANSI-enhancements implemented.

1.2 Chapter Overview

This manual is not a primer and it has been written to complement other documentation. The objective of the C-chapter is to explain the specifics of using the Archimedes C Cross-compiler for microcontroller development relative to a standard native C-compiler, rather than exploring the basics of the C-language.

In other words, this manual builds on any earlier C-experience you might have. You are recommended to have your favorite C book handy as a general C language reference. As a tutorial, *The C Programming Language*, by the original C authors Kernighan &

Ritchie, is suitable though its language definition differs slightly compared to the proposed ANSI-standard. Also, see the Tutorial for other recommended C-literature.

The ANSI-standard documentation is available from the ANSI committee. However, it is not recommended reading since it is written with the sole purpose of clearly defining a standard. Sections 18.1 and 18.2 provide a quick reference to the most important language enhancements through the ANSI-standard.

To fully utilize the capabilities of the Archimedes C-compiler you also need to familiarize yourself with the other chapters in the manual. Appendix B summarizes all the C error and warning messages. You will also find several of the other appendices useful.

1.3 C-6811 Overview

The Archimedes C-compiler is a portable program almost completely written in C. It is based on a memory-intensive design (no temporary files or overlays) that results in a very speedy and also relatively simple system. The compiler executes as a single program that directly generates relocatable code in Archimedes's proprietary UBROF format (Universal Binary Relocatable Object Format).

Several code generating options are available to best support the requirements of different microcontroller designs. Different models support systems using internal RAM only, e.g. "single-chip" applications, as well as systems using external RAM. Also, special models are available to fully support recursive and reentrant functions.

The compiler performs code optimization by default. Further optimization is possible by using the size optimizing switch (-z) or the speed optimizing switch (-s). When optimizing for speed, the compiler increases the in-line code generated and decreases the number of function calls to helper routines thus increasing the execution speed. When optimizing for size, however, the compiler decreases the size of the in-line code and increases the number of calls to helper routines thus decreasing the overall size of the code.

In addition to the type-checks required by the ANSI-standard, Archimedes has also integrated a substantial part of the functions provided by the UNIX C program verifier 'LINT' directly in the compiler. With this facility (activated by the command line switch -g) programs using the older K&R-form of function declarations/definitions can be checked during compilation of a

module (module = file). Unlike LINT which only works at the source-level, the Archimedes system leaves the interface-checking to the linker. This has the advantage of making checks on library function usage feasible, even when only available in object form. Identifiers may have no more than 255 characters through-out the system (compiler, assembler and linker).

The compiler output is to be processed by the Archimedes universal linker XLINK to obtain an executable file. The linker output is usually fed to the target system, PROM-programmer, or emulator.

A C-compiler option of generating assembler code may be used to facilitate debug. It has options of generating full cross-reference lists as well as paginated list-files. The compiler has advanced error recovery and diagnostic systems. Arrows indicate the exact source code error location and an error message describes the error detected.

The Archimedes C-compiler is invoked in a manner similar to a UNIX compiler but with the addition of a simple built-in help facility. C source code can be written with any standard ASCII full-screen editor.

2. Data Representation

The C compiler supports all ANSI C basic elements. Variables are always stored with the most significant part located at low memory address. Variables of type "Float" are implemented in the IEEE 32-bit single precision format. Variables of type "double" and "long double" are implemented in the IEEE 64-bit double precision format. Plain "char" is equivalent to "unsigned char" in this implementation (which can be changed with the -c switch). The table below shows the size in bytes of the different objects.

char	short	int	long	float	pointer/address
1	2	2	4	4/8	2

Note that the ANSI-standard specifies short integer as a 2-byte data-type. If you prefer to have an 1-byte short integer (in your whole program) you can accomplish this by using the C-language #define pre-processing directive and the signed char data type (e.g. #define shrt signed char).

As in any standard C, integer hex constants may be entered in the "0xA800" format. Register variables as a storage class is recognized but currently ignored by the compiler due to the register requirements of the C run-time code.

"Signed char" is an ANSI-standard 1-byte element. It can be used for both characters and numbers range (-127 to +127).

Floating point supports the four basic arithmetic operations (+, -, *, /). The following is a list of some of the advanced math functions that are available: atan, asin, acos, tan, cos, sin, sqrt, exp, pow, exp10, log and log10. To ease the use of the mathematical functions a header file math.h is also featured in the package. Refer to App. F for a more complete list of these functions.

Floating point numbers are stored as follows with the most significant bit to the left (i.e. the "Sign" bit is the most significant bit):

Single precision:

[Sign = 1 bit] [Exponent = 8 bits] [Mantissa = 23 bits]

In other words, the registers A, B and IY (two bytes) contain the following information:

SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM

The "Sign" bit is 1 if it's a negative number. The "Mantissa" is normalized.

A normalized non-zero number X has the form of:

$$X_{\text{float}} = [(-1)^S] \times [2^{(E-127)}] \times [1.F]$$

where,

S = Sign bit

E = 8-bit exponent biased by 127.

F = X's 23-bit fraction which together with an implicit leading 1 yields the significant digit field "1.--".

Double precision:

[Sign = 1 bit] [Exponent = 11 bits] [Mantissa = 52 bits]

$$X_{\text{double}} = [(-1)^S] \times [2^{(E-1023)}] \times [1.F]$$

where,

S: sign bit

E: 11-bit exponent biased by 1023.

F: X's 52-bit fraction.

3. Memory Models

The C-6811 compiler supports several hardware configurations, or memory models, to best meet the requirements of different microcontroller designs. There are six memory models available, selectable with the -m switch (see section about Compiler Switches and Options). The small models make the compiler select the short (direct) addressing mode when accessing static/global variables. The small models apply to systems with no external RAM (i.e. single-chip) while the large and banked models are suitable for systems with RAM expansion.

All memory models can be selected to be either static or reentrant. In the reentrant models all local "auto" variables are allocated and deallocated dynamically on the stack. This process is necessary if recursive or reentrant functions are needed. The reentrant models sometimes generate larger and slower code than the static models. In the static models all function-level variables are put into static memory locations, with the exception of function arguments which are always on the stack.

The different C-6811 memory models are:**Large reentrant memory model (-ml)**

Supports microcontroller applications using external RAM (in addition to the internal RAM of the microcontroller).

All local "auto" variables are allocated and deallocated dynamically on the stack, which is necessary to fully support recursive or reentrant functions. Note that -ml is the default compiler option.

Large static memory model(-ml -d)

Supports microcontroller applications using external RAM (in addition to the internal RAM of the microcontroller). All function-level variables are put into static memory locations, with the exception of function arguments which are always on the stack.

Small reentrant memory model (-ms)

Supports microcontroller applications using internal RAM only (single-chip applications). All local "auto" variables are allocated and deallocated dynamically, on the stack.

Small static memory model (-ms -d)

Supports microcontroller applications using internal RAM only (single-chip applications). All function-level variables are put into static memory, with the exception of function arguments which are always on the stack.

Banked reentrant memory model (-mb)

Supports microcontroller applications using more than 64K of code. . All local "auto" variables are allocated and deallocated dynamically on the stack.

Banked static memory model (-mb -d)

Supports microcontroller applications using more than 64K of code. All function-level variables are put into static memory, with the exception of function arguments which are always on the stack.

Code from all models may be partially tested on the host computer using resident C compilers and debuggers. This approach assumes that you follow common ANSI C rules for writing machine-independent code. Of course, code with functions relying on 68HC11

I/O specific parts may not be verified in this way. The following table shows what memory model to select for each hardware configuration:

Memory Model	Banked (-mb/-d)	Large (-ml/-d)	Small (-ms/-d)
External RAM	Yes	Yes	No
Code Area	>1M	64K	64K
Variable Area	<64K	<64K	<256
C Library	cl6811b*.r07	cl6811*.r07	cl6811*.r07

Note that only one memory model can be used in a given application. To best choose the model for your application, consider the following aspects:

- Internal vs. external RAM.
- The size of local variables
- Use recursion, function call nesting, printf, sprintf, malloc and free library functions.
- Time-critical code.
- Recursive and reentrant code.

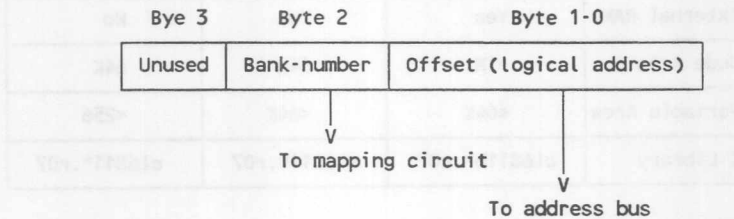
If a statically allocated object is prefixed with "const" this object will be put into the ROM section only (CONST segment). This is valuable when a large number of static data structures must be created. This is an ANSI supported feature.

```
typedef struct
{
    declarators.....
} s;
const s table[] =
{
    initializers.....
};
```

The "banked" model is identical to the "large" model in terms of variable allocation and initialization. However, the code area can be extended (transparently at the C level) with up to 256 64K-blocks of memory. However, one block (root block) must be accessible at all times.

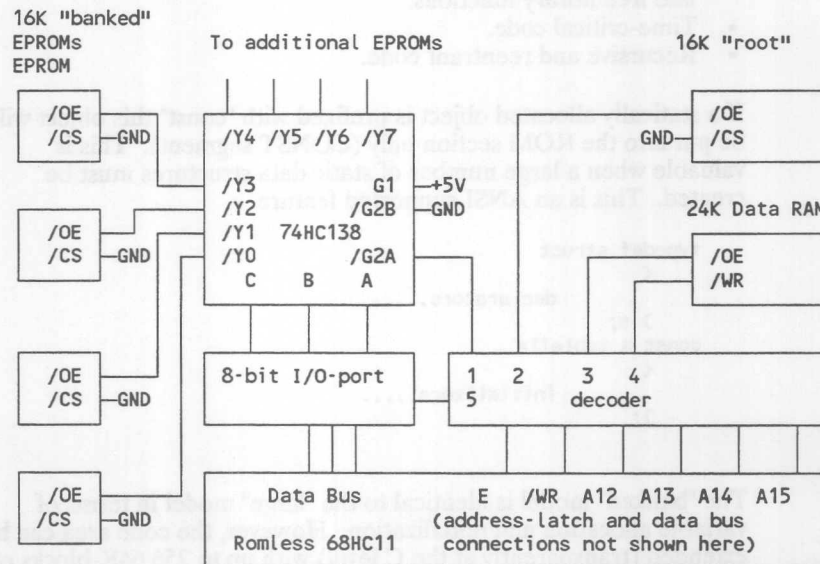
To overcome the fact that function addresses are still 16 bits wide, all non-local function calls are made indirectly through a 32-bit static function address. That is: A global function address does not point to the function, only to its indirect address location.

In a picture:



To accomplish this, the compiler generates static function pointers (one for each non-local function definition in a module) and allocates them to a segment called FLIST.

A typical PROM-based system could be configured like this:



Where the decoder functions are:

1. $\text{/E} + \text{WR} + \text{/A15} + \text{A14}$ (8000 - BFFF)
2. $\text{/E} + \text{WR} + \text{/A15} + \text{/A14}$ (C000 - FFFF)
3. $\text{/E} + \text{A15} + \text{WR} + \text{/(A14} + \text{A13)}$ (2000 - 7FFF)
4. $\text{/E} + \text{A15} + \text{/WR} + \text{/(A14} + \text{A13)}$ (2000 - 7FFF)
5. $\text{/E} + \text{A15} + \text{A14} + \text{A13} + \text{/A12}$ (1000 - 1FFF)

This system utilizes up to eight 16K banks to create a system with 144K of code. Port #1 was "sacrificed" for the purpose of mapping the actual bank to execute. Other ports may be used as shown in file l09.s07 which contains the actual switching routines (user-configurable for other mapping hardware/schemes).

In this system the "root" EPROM is allocated to address C000-FFFF (hex) which is most practical since 68HC11 programs fetch data at location FFFE after reset. The root memory contains all vital intrinsic library routines (i.e. C support code like floating point arithmetic, rather than user-callable functions) needed by the C program. Due to the 68HC11 C run-time requirements all constant data must also be located in the root memory.

The data RAM (located at 0000-7FFF), contains stack and static variables. The "banked" memory blocks start at logical address 8000 (hex) which means that the -b flag of the linker should be set as follows: -bCODE=8000,4000,10000.

The first parameter is the logical address of bank 0 (Port #1 = 0) and the second parameter specifies the bank size (16K banks in this system), while the third parameter specifies the increment in bank numbers (e.g. bank-number 0, 1, 2, 3 etc). Only executable code of callable functions can reside in the banked area. However, interrupt handlers, C runtime routines, compiler support routines, and constant data objects must reside in the root bank.

IMPORTANT

No single module can be larger than the bank-size (over-sized modules result in linker error-messages).

It is recommended to keep module size considerably smaller than bank-size in order to avoid memory fragmentation (partially filled banks), as the linker only packs complete modules into banks.

The compiler will, in the banked mode, select the fast direct calling method (same as in the non-banked modes) if a function is considered as local (i.e. has the storage-class static and is not referenced as a function pointer), or is declared with the non-banked keyword.

4. Absolute Read/Write at C-level

It is possible to access specific memory locations directly from a C-program, thus making it possible to read/write to memory mapped I/O peripheral devices, without having to write special assembly routines for it. The following example shows the technique.

```
#define PORT (*(char *) 0x4000)    /* Pointer to address 4000 (hex) */

void read_write (char c)
{
    PORT = c;                      /* Write c to absolute address */
    c = PORT;                      /* Read from absolute address */
}
```

To ease the use of 68HC11 on-chip I/O features, a special include file, io6811.h, containing the most important address definitions, is also supplied in the compiler kit.

5. Register Usage

All compiler memory models use all registers (A, B, IX, IY, and CCR), which is important to know when writing assembly language support routines or interrupt handlers. The registers are used as follows:

A, B, CCR, IX, IY

Hold temporary results and do not require saving in assembly language routines.

A, B

Hold function return values for all types except "long" and "float" (structures are always returned as addresses). A contains the most significant part of the result for 16-bit results while B holds the entire value in case of "char" return values.

A, B, IY

Hold function return values for "long" and "float" return values. A,B contain the most significant part of the result while IY contains the least significant part.

6. Assembly Language Interface

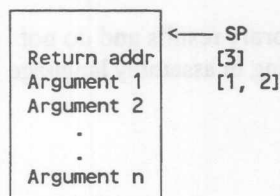
Assembly language routines may be useful in optimizing time-critical sections of code. This requirement typically applies to small portions (2-10%) of a program. Understanding the execution profile of the completed system is, therefore, important in time critical applications. The following sections describe the interface between the C system and the assembly routines. Calling C routines from assembly is also possible but has few applications. For more detailed examples, study the compiler's output using the -q switch.

7. Parameter Stack

Parameters are pushed on the stack in reversed order with the return address as top element. Immediately after a call to a function, the stack contains the following:

B/A:B_reg = Argument 1 [1, 2]

low memory



high memory

- [1] Banked model: A:B contents are undefined, all function arguments reside on stack.

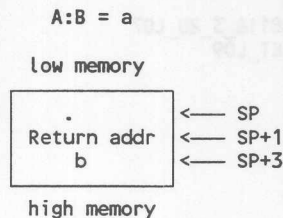
Non-banked models: First function argument resides in register B (8-bit) or register A:B if greater than 8-bit with one important exception: "struct" and "union" objects are always pushed on the stack completely before a function call is performed. In case of "long" and "float" data types the least significant word is pushed on the stack while the other part is in A:B.

- [2] A special technique is used for "struct" and "union" return values. The caller reserves an area somewhere in its own "auto" space and gives the called function an address to that area as a first parameter (in A:B). The called function is in this case responsible for using that parameter as a return value location as well as setting register A:B to that address at the time of return. In the banked model the caller pushes the return value address before calling the function.
- [3] The return address will, in the banked model, also contain a byte holding bank-number of the invoking function. This byte is the last pushed object at the time of a function call.

A function return value should be stored in A:B:IY if it is 32-bit, A:B if it is 16-bit, and in B if it is 8-bit, when returning to the caller. "void" function return values leaves contents of A:B undefined. In the case of 64-bit return values, the calling function passes the called function a pointer to a storage memory space where the called function is expected to place the return value. The storage memory space must be in the calling function's local space.

The called function is always responsible for 'popping' all own variables from the stack before returning to the caller, while the caller is responsible for popping pushed arguments

Now assume that a routine unsigned long uadd (unsigned int a, unsigned int b) is to be written. A non-banked call to this function from a C program will result in the following stack:



The function may then be accomplished with the following code:

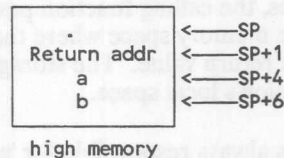
```

NAME      tst(16)
RSEG      CODE(0)
PUBLIC    uadd
EXTERN    ?CL6811_3_20_L07
RSEG      CODE

uadd:
P68H11
PSHB
PSHA
TSX
LDD       0,X
ADDD      4,X
PSHB
PSHA
CLRA
CLRB
PULY
PULX
RTS
END

```

A bank-switched call to uadd would result in the following parameter stack:



The function could be implemented with this code:

```

NAME      tst(17)
RSEG      CODE(0)
RSEG      FLIST(0)
PUBLIC    uadd
EXTERN    ?CL6811B_3_20_L07
EXTERN    ?X_RET_L09
RSEG      CODE
?0000:
P68H11
TSX
LDD       3,X
ADD       5,X
PSHB
PSHA
CLRA
CLRB
PULY
JMP       ?X_RET_L09
RSEG      FLIST
uadd:
FQB       ?0000
END

```

In this case one of the two routines (defined in l09.s07) controlling the bank-switch circuitry, was used. That is, a return from function must also restore the bank-number.

8. Linking

Linkage of object modules created by the 68HC11 C-compiler requires several steps that are best carried out by using a linker command file (xlink -f file). To ease the creation of such files, a framework is supplied in the form of files lnk6811.xcl and lnk6811b.xcl for the non-banked and banked models, respectively. The contents of lnk6811.xcl is shown below:

```
-! -LNK6811.XCL-

XLINK 4.xx command file to be used with the 68HC11 C-compiler V3.x
using the -ms or -ml options (non banked memory models).
Usage: xlink your_file(s) -f lnk6811

First define CPU -!

-c68hc11

-! Allocate segments which should be loaded -!

-! First allocate the read only segments.
    6000 was here supposed to be start of PROM -!

-Z(CODE)RCODE, CODE, CDATA, ZVECT, CONST, CSTR, CCSTR=6000

-! The interrupt vectors are assumed to start at FFD6, see also
    CSTARTUP.S07 and INT6811.H.

-Z(DATA)INTVEC=FFD6

-! Then the writeable segments which must be mapped to a RAM area
    2000 was here supposed to be start of RAM.
    Note: Stack size is set to 128 (80H) bytes with 'CSTACK+80'
    !

-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK+80=2000

-! NOTE: In case of a RAM-only system, the two segment lists may be
    connected to allocate a contiguous memory space. I.e. :
    -Z...CCSTR, DATA...=start_of_RAM -!

-! The segment SHORTAD is for direct addressing, let XLINK check
    that the variables really are within the zero page -!

-Z(DATA)SHORTAD=00-FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write

-! See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read

-! Now load the 'C' library -!
cl6811 -! or cl6811d -!

-! Code will now reside in file aout.a07 in MOTOROLA 'S' format -!
```

The resulting code is downloaded to an emulator, single-board computer or PROM-programmer. Other formats than the default can be set with the -F option.

The supplied linker command file contain several segments defined for the C run-time and the PROMming of code. Although your particular program might not be using all of these segments, you are recommended to leave these segment definitions in your linking file as you do not gain anything by removing a segment. The defined segments in the linker command file are (upper case is significant):

ROM Segments

RCODE

Used by the CSTARTUP routine and routines called by the C code generator. Banked mode note: also suitable for user-written assembler code that is not called from C (interrupt handlers and similar resident code).

CODE

Used for C library routines as well as user-written routines and is also suitable for holding code for assembly language routines. Banked mode note: should only hold code that is callable from C (i.e. bank-switched code).

FLIST

Only generated in the banked mode (also see section 3).

CDATA, ZVECT, CSTR, CCSTR, CONST

Are used for initializing C variables at startup and may not be used in a user program.

INTVEC

Used to hold the interrupt vector table entries (Jumps to ISRs).

RAM Segments

DATA

Used for allocating variables (except when PROMable code is generated). May also be used in assembly language routines.

UDATA

Used for allocating variables that should be set to zero at startup. May also be used in assembly language routines.

IDATA, WCSTR, ECSTR

Are used for initializing C variables at startup and may not be used in a user program (see CSTARTUP.S07).

CSTACK

Holds the run-time stack segment (size = stackspace).

TEMP

Used to hold auto variables in the static memory models.

NO_INIT

Used to hold data objects that should not be initialized by the startup code. It holds variables declared with the no_init keyword.

SHORTAD

Used to hold zpage data objects declared with the zpage keyword.

Segment names, other than those mentioned above, may be used and can be placed anywhere in the applicable segment list (see section 8. linking), or mapped to a special area like an I/O device or battery-powered RAM.

9. Configuration Issues

If the hardware must be initiated before the C program starts, code can be added to the file `cstartup.s07` which contains the initialization routine (in assembly source). Control must be transferred to the first address of the `CSTART` segment (only used in `cstartup.s07`).

Due to the fact that the banked and non-banked memory models are incompatible, the user-configurable as well as user-written assembly language routines must be tailored for a particular model. To make assembly routines more general, assembly routines supplied with this package reference a file called `defcpu.inc`. This file is created by copying one of the files `def6811.inc` (non-banked) and `def6811b.inc` (banked) to `defcpu.inc`. Conditional assembly can then be applied to a routine. For more details, study `cstartup.s07`.

After modifications, `cstartup.s07` should be re-assembled and the corresponding object file `cstartup.r07` should replace the original startup routine. This module replacement is best accomplished by using the following sequence with the library manager program:

```
C>xlib
*def-cpu 68HC11
*rep-mod cstartup cl6811
*exit
```

The `printf` and `sprintf` routines share a common formatter which is pre-installed in the library as module `_formatted_write`. Due to the code size and run time stack size required by this full ANSI version of the formatter, three different formatters are actually installed in the C library files. These are:

`_formatted_write`: Supports full ANSI `printf/sprintf` definition.

`medium_write`: Differs from the ANSI version by not supporting floating point numbers. It is about half the size of the full formatter.

`small_write`: A small version of the formatter that only supports the basic `%i`, `%d`, `%o`, `%c`, `%s`, and `%x` specifiers for integer type objects. It does not support field width or precision arguments. The size of this routine is about 20% of the `_formatted_write`.

The `-e` switch should be used at link time if a smaller version of the formatter is desired. For example, to link the smallest version of the `_formatted_write`, the following line must be included in the linker command file:

```
-e_small_write=_formatted_write
```

For details on how to modify the output from `putchar` and `printf` please refer to file `putchar.s07`.

Similarly, the `scanf` and `sscanf` routines share a common formatter called `_formatted_read`. Due to the code size and the run time stack size required by the full ANSI version of the formatter, a smaller version of the formatter is installed in the C library files. The two versions are:

`_formatted_read`: Supports the full ANSI `scanf/sscanf` definition.

`medium_read`: Differs from the ANSI version by not supporting floating point numbers.

The `-e` linker switch must be used if the smaller version of the formatter is desired. For example, to link the medium version of the `_formatted_read`, the following line must be included in the linker command file:

```
-e_medium_read=_formatted_read
```

Note that if a specifier is used with a formatter version that does not support that particular specifier, the specifier itself will be printed out in the case of `printf/sprintf` or `read` in the case of `scanf/sscanf`.

10. Stack and Heap Size

By default the run-time stack size is set to 0 bytes. This desired size of the stack can be set at link-time by adding `+value` to the `CSTACK` segment:

```
-ZSEGA,SEGB,CSTACK+value=address
```

The stack requires, as a general guideline, 2 bytes per level of subroutine nesting, 10 bytes for an integer operation, 20 bytes for a float operation, about 20 bytes at any time for C run-time library functions and at least 25 bytes for interrupt support.

The heap is only needed (and loaded) if the library routine `malloc` is called. The preset heap size is 2000 bytes. If the size of this area must be changed, the file `heap.c` should be recompiled and re-installed in the C library again using `XLIB`.

11. Interrupt Routines

Interrupts are handled in one of two ways: C or assembly service routines. In the case where interrupt service routines are to be written in C, the "interrupt" keyword can be used to fill up the interrupt vector table automatically without having to modify the CSTARTUP routine.

On the other hand, if the interrupt service routines are to be written in assembly, the interrupt vector table must be filled out manually in the CSTARTUP.S07 file. This file must then be assembled and replaced in the C library file used in that particular application. Refer to the librarian section of this manual for more details on module replacement.

Note that since the processor saves all registers at the time of an interrupt acknowledge, interrupt handlers written in C or assembler do not have to save any register.

Examples on the Use of Interrupts

Below are two examples on how to write interrupt service routines in both C and assembly.

ISRs Written in C:

```
interrupt [vector] void intrpt_srvc_rtn(void); /* ANSI C prototype */

interrupt [vector] void intrpt_srvc_rtn(void)
{

/* ISR definition */

}
```

For more information of the interrupt keyword, refer to section 1.14 on C-6811-specific extensions.

ISRs Written in Assembly:

The following code must be added to the CSTARTUP routine for each interrupt service routine:

```

                                EXTERN  intrpt_srvc_rtn1
                                RSEG    RCODE
startup:
                                LDS      #stack_begin-1
                                :
                                :
                                JMP     ?C_EXIT
handler1:
                                JSR      intrpt_srvc_rtn1
                                RTI

                                COMMON  INTVEC          Linked at FFD6
                                RMB      28
                                FDB     handler1         located at FFF2

```

12. 68HC11-Specific Extensions**12.1 In-line Functions**

In addition to the routines listed in the general C library section (Appendix B), the `-e` (extension) option forces the compiler to recognize several pre-defined in-line functions for controlling the 68HC11 interrupt system. This feature coupled with absolute read/write (see section 1.6) minimizes the need for assembly language routines considerably. The in-line functions can be used in the same contexts as ordinary functions with one exception: they cannot be referred to as addresses in initializers and in function pointer assignments. In addition they behave like reserved words. The in-line functions conceptually work like the following ANSI-type function declarations:

enable_interrupt

```
void enable_interrupt(void);
```

Turn on interrupts (CLI).

disable_interrupt

```
void disable_interrupt(void);
```

Turn off interrupts (SEI).

wait_for_interrupt

void wait_for_interrupt(void);
 Executes the WAIT-instructions.

12.2 Extended Language Keywords**Zero Page Memory Support**

To facilitate the declaration of variables in the lowest addressable part (0-FF) of the 68HC11, a new keyword "zpage" was added. This can be used to put more frequently used variables in the memory where they can be accessed with the efficient direct 8-bit addressing mode.

Declaration syntax:

```
zpage declarator
storage-class opt zpage declarator
```

Variables of the type zpage cannot be initialized at compile-time and may only be declared at file-level.

Examples:

```
zpage int buf[10];
static zpage char quick;
zpage char *p = "str";    /* ILLEGAL: initializer */
```

The zpage variables will be generated on the segment SHORTAD which should be linked to begin at RAM address zero:

```
-ZSHORTAD=0
```

Also see the *memory* #pragma later in this chapter.

Non-volatile Memory Support

To support non-volatile memory there is a special attribute "no_init", which can be used to specify variables that are not initialized at startup (as other static C variables). These variables are loaded into a special segment called NO_INIT, and located to the start address of the non-volatile memory (with the -Z option of XLINK).

Declaration syntax:

```
no_init declarator
storage-classopt no_init declarator
```

Variables of the type `no_init` cannot be initialized at compile-time and may only be declared at file-level.

Examples:

```
no_init int settings[50];
static no_init char configured;
no_init char *p = "str"; /* ILLEGAL: initializer */
```

Assuming that a non-volatile RAM is located at address 8000 (hex) the required additional command to XLINK should read:

```
-ZNO_INIT=8000
```

'no_init' declarations can be used for the built-in EEPROM, but the EEPROM requires an external software driver for write-operations (which is currently not supplied in the package).

Also the *memory* #pragma later in this chapter.

Interrupt Functions

With the extension keyword "interrupt", it is possible to write interrupt handlers directly in C as the compiler exclusively generates the code in the vector table.

Declaration syntax:

```
interrupt function-declarator
interrupt [vector]opt function-declarator
storage-classopt interrupt function-declarator
storage-classopt interrupt [vector]opt function-declarator
```

If a [vector] constant expression is given, an interrupt vector pointing to the interrupt function will be generated and inserted at absolute address (INTVEC + vector value) in the memory. An interrupt function will preserve register contents and return with an RTI instead of the usual RTS. If no [vector] is given the user must provide a proper vector to the interrupt function (preferably in the CSTARTUP module). Forward declarations with [vector] numbers will transfer the vector information to subsequent function

definitions. This feature is used in the 68HC11-specific INT6811.H file which contains predefined interrupt declarations. Interrupt functions must be void and have no arguments.

Examples:

```
interrupt [0] void SCI_int(void) /* SCI interrupt handler */
{
    PORTA = 6;
}

interrupt [8] void TO_int(void); /* Only a declaration */

interrupt void TO_int(void) /* Timer 0 handler */
{
    if (PORTA & 1) start_engine();
}
```

Note that the compiler does not allow direct calls to interrupt functions, but allows interrupt function addresses to be passed to function pointers (that cannot have the interrupt attribute). The latter can be used to 'install' interrupt handlers in operating system environments.

In the banked mode, it is highly recommended to keep interrupt handlers in a special file, compiled with option -RRCODE, and separated from other user functions, as the calls to the handlers are non-banked (i.e. requires placement in non-banked memory). Also see the function #pragma in this chapter, as well as the code example illustrating the use of an interrupt handler supporting banking.

Monitor Functions

With the extension keyword "monitor", functions can be made to execute with the interrupt system disabled. This is used to facilitate "atomic" operations (like semaphores in operating systems). After return from a monitor function the interrupt system will be restored to the state it had at the time of the call. Monitor functions can be used in the same contexts as ordinary functions (i.e. same parameter passing etc.).

Declaration syntax:

```
monitor function-declarator
storage-classopt monitor function-declarator
```


Examples:

```

char printer_free;
monitor int got_flag(char *flag)
{
    if (!*flag)                /* ATOMIC operation */
    {
        return (*flag = 1);    /* SUCCESS */
    }
    return (0);                /* WAS NOT FREE */
}

void f(void)
{
    if (got_flag(&printer_free))
        .... action code ....
}

```

The above shows a typical usage of the monitor function concept.

Non-banked Functions

In the banked mode, global functions always use the banked calling mechanism. This overhead is often not desired, particularly when external assembly language routines residing in non-banked memory are to be linked and called from C. With the extension keyword "non_banked", it is possible to force a function which normally would require banked calls to only be called with standard JSRs. The non_banked keyword can also be applied to any C function provided that the declaration is consistent between the modules involved (checked by XLINK if they are compiled with the -g option), and that the caller and called function is really residing in the same bank. The latter requires that a set of modules that belong together are compiled with the -R option to collect executable code under a common segment name.

Declaration syntax:

```

non_banked function-declarator
storage-class_opt non_banked function-declarator

```

Examples:

```
extern non_banked void asm_sub(void); /* External non-banked
function */
non_banked void nb(void)             /* Global non-banked function */
{
}
```

Also see the *function* #pragma later in this chapter.

#pragma Commands

With the ANSI-specified #pragma directive it is possible to add new functionality to the language in a controlled way. Note that #pragma directives can be executed whether the -e option is on or not. C-6811 3.10 supports the following #pragma keywords:

language #pragma

```
#pragma language={ extended | default }
```

To activate the extended keywords described earlier, you can either specify the -e compiler switch or insert the line

```
#pragma language=extended
```

in the source code. To restore the language level to the state it had at invocation time (-e option dependent) specify:

```
#pragma language=default
```

memory #pragma

```
#pragma memory={ zpage | no_init | dataseg | constseg | default }
```

To ease the process of declaring variables to reside in other memory segments than the default, the memory #pragma offers a 'cleaner' alternative to the memory attributes described earlier. Note that the memory #pragma can be overridden by the memory attributes.

Example:

```
#pragma memory=zpage

int buffer[10];    /* 0-FF memory */
extern double d;    /* 0-FF memory */
no_init char *strings[5]; /* Overrides to no_init memory */

#pragma memory=default

int i;             /* Default memory type */
```

constseg, dataseg #pragma

User-defined segments are now allowed if used as shown in the following example.

Example:

```
#pragma memory=dataseg(USART)
char USART_data;
char USART_control;
int USART_rate;
#pragma memory=constseg(TABLE)
char arr [ ] = { 6,9,2,-5,0 };
#pragma memory=default
```

The following "extern" declaration must be used when referring to these symbols from other modules:

```
#pragma memory=dataseg(USART)
extern char USART_data;
.
.
etc
```

"dataseg" is equivalent to the "no_init" type while "constseg" is equivalent to the "const" type.

Note that a non-default memory #pragma will generate error-messages if function declarators are encountered.

function #pragma

```
#pragma function={ non_banked | interrupt | monitor | default }
```

With the function #pragma any number of function definitions and declarations can be set to the same type as with the non_banked, monitor, and interrupt function modifiers (see sections concerning interrupt, monitor and non_banked keywords).

Examples:

```
#pragma function=non_banked

extern void f1(void); /* Identical to extern non_banked void f1() */

void f2(void) /* Will make this function non_banked */
{
}

#pragma function=default

extern int f3(void); /* Default function type */
```

Note

#pragma function=interrupt does not offer a vector option (it can, however, inherit earlier interrupt [vector] function declarations).

#pragma codeseg(NAME)

This #pragma sets the code segment to NAME. Its effect is similar to that of the -R option. Note, however, that this #pragma can be executed only once. It overrides the effect of the -R option. The segment NAME must not conflict with data-segments.

#pragma warnings

This #pragma allows the user to control the status of the warning messages.

Example:

```
#pragma warnings=on      /* turns warnings ON */
#pragma warnings=off     /* turns warnings OFF */
#pragma warnings=default /* enables the -w option */
```

#pragma bitfields

This #pragma reverses the order in which bitfields are allocated in memory. The default setting is used, bitfields are allocated LSB at higher address and MSB at lower address. When the #pragma bitfields=reversed is used, LSB will be located at lower address and MSB will be located at higher address.

Syntax

```
#pragma bitfields={reversed | default}
```

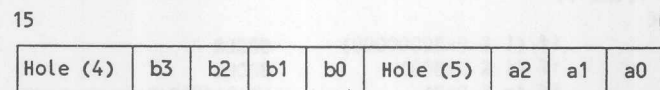
Example:

The default memory allocation for the following structure is:

Definition:

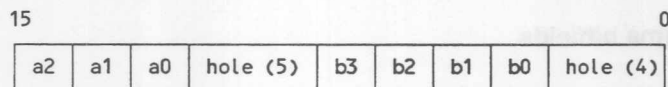
```
struct
{
    unsigned short a:3;
    unsigned short :5; /* Hole */
    unsigned short b:4;
} var;
```

Memory allocation:



Definition:

```
#pragma bitfields=reversed
struct
{
    unsigned short a:3;
    unsigned short :5; /* Hole */
    unsigned short b:4;
} var;
#pragma bitfields=default
```

Memory allocation:**BRCLR/BRSET Instruction Usage**

The compiler uses the BECLR and BRSET instructions when:

1. Using conditional statements such as if, while, do-while, &&, ||.
2. The operation of the type "var & mask" or "!(var & mask)"
 - a. The tested mask value is a byte (BRCLR) or a bit (BRSET).
 - b. "var" is a zero-page variable (relocatable or constant I/O pointer), or an auto variable.
3. The target label can be reached with $a \pm 128$ relative offset.

Examples:

```
char c;
zpage long l;
void f(int i)
{
    if (l & 0x80000000)    BRCLR
    if (i & 0x800)         BRCLR
    if (c & 0x8)           LDAB+ANDB+Bcc -> not Zero-page
    if (l & 0x00F300)      BRCLR
    if (!(l & 0x0800))     BRSET
    if (!(l & 0xF300))     LDAB+ANDB+Bcc -> not one bit
}
```

Combination of "non_banked" & "monitor" Keywords

The C-6811 compiler allows the declaration of functions that are simultaneously *non-banked* and *monitor*, as shown :

```
non_banked monitor int f(int i)
{
}
```

Primitive In-Line Assembler

```
void _opc (a_constant);
```

When the "-e" option is activated the compiler recognizes a special pre-declared function "_opc" that takes a constant as an argument and puts out that argument as an FCB value in the position where it stands.

Example:

```
void stop_and_return_to_OS()
{
    _opc (0x4F); /* CLRA */
    _opc (0x7E); _opc (0x20); _opc (0x00); /* JMP $2000 */
}
```

12.3 Additional Examples

Below are examples showing the most important new C language constructs:

```
/*=====*/
/* Direct interrupt support */
/*=====*/
#define PORT (*(char *)0x1000)

interrupt [32] void SWI_interrupt(void); /* Forward (usually in
int6811.h) */

interrupt void no_vector_gen_interrupt(void) /* User creates vector
in ASM */
{
    PORT = 2;
}
```



```

interrupt void SWI_interrupt(void)          /* Note that vector is
known */
{
    PORT = 1;
}

/*=====*/
/* Zero page support gives shorter code for important variables */
/*=====*/
zpage int fast_var;

/*=====*/
/* Non-initialized memory option supports battery-backed RAM etc*/
/*=====*/
no_init int backed_var;

void main(void)
{
    fast_var = 1;
    backed_var = 2;
}

```

The previous program generates the following code:

```

\ 0000 NAME news6811(16)
\ 0000 RSEG CODE(0)
\ 0000 COMMON INTVEC(0)
\ 0000 RSEG SHORTAD(0)
\ 0000 RSEG NO_INIT(0)
\ 0000 PUBLIC SWI_interrupt
\ 0000 PUBLIC backed_var
\ 0000 PUBLIC fast_var
\ 0000 PUBLIC main
\ 0000 PUBLIC no_vector_gen_interrupt
\ 0000 EXTERN ?CL6811_3_10_L07
\ 0000 RSEG CODE
1 /*=====*/
2 /* Direct interrupt support */
3 /*=====*/
4 #define PORT (*(char *)0x1000)

5 /* Forward (usually in int6811.h) */
6 interrupt [32] void SWI_interrupt(void);

7 /* User creates vector in ASM */
8 interrupt void no_vector_gen_interrupt(void)
9 {
\ 0000 no_vector_gen_interrupt:
10 PORT = 2;
\ 0000 C602 LDAB #2
\ 0002 F71000 STAB 4096
11 }
\ 0005 3B RTI
12 /* Note that vector is known */
13 interrupt void SWI_interrupt(void)

```

```

14      \      0006      {      SWI_interrupt:
15      \      15      PORT = 1;
16      \      0006      C601      LDAB      #1
17      \      0008      F71000      STAB      4096
18      \      16      }
19      \      000B      3B      RTI
20      \      17
21      \      18
22      /*=====*/
23      19      /* Zero page support gives shorter code
24      for important variables      */
25      20
26      /*=====*/
27      21      zpage int fast_var;
28      22
29      23
30      /*=====*/
31      24      /* Non-initialized memory option supports
32      battery-backed RAM etc.      */
33      25
34      /*=====*/
35      26      no_init int backed_var;
36      27
37      28      void main(void)
38      29      {
39      \      000C      main:
40      \      30      fast_var = 1;
41      \      000C      CC0001      LDD      #1
42      \      000F      DD00      STD      <fast_var
43      31      backed_var = 2;
44      \      0011      CC0002      LDD      #2
45      \      0014      FD0000      STD      backed_var
46      32      }
47      \      0017      39      RTS
48      \      0000      COMMON      INTVEC
49      \      0020      RMB      32
50      \      0020      0006      FDB      SWI_interrupt
51      \      0000      RSEG      SHORTAD
52      \      0000      fast_var:
53      \      0002      RMB      2
54      \      0000      RSEG      NO_INIT
55      \      0000      backed_var:
56      \      0002      RMB      2
57      \      0002      END

```

The resulting code adds the following segments:

INTVEC: Contains the generated interrupt vectors and is supposed to be located at the start of the interrupt vector memory section for the target.

NO_INIT: Contains variables that are supposed to be located at the place of a non-volatile RAM.

SHORTAD: Contains zero-page variables that should reside in address 0-FF.

The additional commands to XLINK would then typically look like this:

```
-ZINTVEC=FFD6
-ZSHORTAD=0
-ZNO_INIT=place_of_the_non_volatile_RAM
```

Note: If zpage variables are used, it is necessary to also modify the current RAM allocation for the data segments. For example, if one int variable is specified as zpage the given segment list:

```
-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK=0
```

would be modified to:

```
-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK=2
```

To set and clear individual bits as well as groups of bits in the I/O registers, the C lines:

```
PORT |= 4;          /* Sets PORT bit 2 */
PORT &= 0xF1;       /* Clears PORT bit 1-3 */
```

are recommended since the compiler generates very efficient code for these constructs (actually only one or two instructions depending on whether the address is direct or extended).

An example interrupt handler that supports banking is presented here :

```
EXTERN heater_overflow
* Here our C program starts to execute
RSEG RCODE
startup:
LDS    #stack_begin-1
.
.
.
handler:
IF     banking
      LDX    #heater_overflow
      JSR    ?X_CALL_L07          Banked "JSR"
ELSE
      JSR    heater_overflow
```

```
ENDIF  
RTI
```

```
COMMON INTVEC  
RMB      34      Interrupt vector (IRQ1)  
FDB      handler
```

13. Operating Instructions

C-6811 [options] sourcefile [options]

The Archimedes C-compiler is a single executable file that is invoked by issuing the name of the compiler program, "C-6811", followed by a source filename with the default extension .C. The source file can optionally be preceded and/or succeeded by compiler option switches recognized by a hyphen character (-). Source filename and compiler switches must be separated by tabs or spaces only. Additional command line input can also be given through a target dependent 'environment' variable and for very complex invocations through a command file (with the -f switch). The latter offsets the current 128 character limit imposed by MS-DOS. If C-6811 is invoked without any options given, C-6811 <CR>, it lists all the options available (=help command).

14. Files

The C-compiler processes one source file at each invocation and the generated code is written on an object file which by default has the basename of the source file plus the 6811-specific (.r07) extension. The basename is the name of a file without directory information and filename extension (i.e. the basename of \usr\myfile.c is myfile). If no object file is wanted, specify: -o nul which forces the compiler to write object code on the MS-DOS 'null' file.

Filenames can be given in any mixture of upper and lowercase letters while command line switches (options) are interpreted "as is". Default file extensions can always be overridden if needed (i.e. myfile.g specified as source file will force the compiler to open myfile.g while myfile will be treated as myfile.c). In addition to the object file, the compiler can optionally produce a list file (-L or -l switches) and/or a file with symbolic assembly code that can be processed by the assembler (-A or -a switches). No object file will be generated if errors are detected during compilation. However, if only warnings occur, an object file will be created.

15. Compiler Switches and Options

The compilation process can be controlled by an extensive set of command line switches/options. The position of the options in the command line is of no importance except for -I commands. Note that all switch commands with a file argument must have the filename separated from the switch flag with tabs or spaces. All other type of arguments should start immediately after the switch character. The table below summarizes all the options.

Archimedes 68HC11 C-Compiler V3.30A/DXT
(c) Copyright Archimedes 1992

```
Usage:      C-6811 {<options>} <sourcefile> {<options>}
Sourcefile: 'C' source file with default extension: .c
Environment: QCC6811
Options (specified order is of no importance):
-o file      Put object on: <file> <.r07>
-Oprefix     Put object on: <prefix> <source> <.r07>
-b          Make object a library module
-P          Generate PROMable code
-g{OA}      Enable global typecheck
              O: Disable object code type-info
              A: Depreciate K&R-style functions
-ms         Select memory model: small
-mL         Select memory model: large (default)
-mb         Select memory model: banked
-d          Force 'static' allocation of 'auto' variables
-2          Set double float to 64 bits
-w          Disable warnings
-s          Optimize for speed
-z          Optimize for size
-y          Put "strings" into variable section
-c          Make plain 'char' = 'signed char'
-e          Enable processor dependent extensions
-f file     Extend command line with <file> <.xcl>
-r{012in}   Enable debugger output in object
              012: Select debug model (0 default)
              i:  Enable #include file source
              n:  No source code option
-l file     Generate a list on: <file> <.lst>
-Lprefix    Generate a list on: <prefix> <source> <.lst>
-tn         Set tab spacing between 2 and 9 (default 8)
-x{DFT2}    Generate cross-reference list
              D: Show all #defines, F: Show all functions
              T: Show all typedefs, 2: Dual line space listing
-q          Put mnemonics in the list
-T          List 'active' lines only (#if etc. true)
-i          List #included files
-pnn        Page listing with 'nn' lines/page (10-150)
-F          Generate formfeed after each listed function
-a file     Generate ASM on: <file> <.s07>
-Aprefix    Generate ASM on: <prefix> <source> <.s07>
```

-Hname	Set object module header = 'name'
-Rname	Set code segment = 'name'
-DSYMB	Equivalent to: #define SYMB 1
-DSYMB=xx	Equivalent to: #define SYMB xx
-USYMB	Equivalent to: #undef SYMB
-Iprefix	Add #include search prefix
-G	Open standard input as source
-S	Silent operation of compiler
-C	Enable nested comments
-K	Enable C++ comments

This summary is also presented on the screen by typing "C-6811" followed by <CR>. All options are now described in more detail. CASE IS SIGNIFICANT IN COMPILER SWITCH OPTIONS.

-ofile Put object in: <file> <.r07>
-Oprefix Put object in: <prefix> <source> <.r07>

By default the compiler will generate object code in a file which name is the basename of the invoking source file plus the 68HC11 specific extension (.r07). With the -o switch you can select the name of the file in which the compiler puts the code. Sometimes it is convenient to have source files in one directory, object in another directory and compile from a third directory. In these cases the -Oprefix can be used to redirect object files from the default (current) directory where they usually are created, to user specified directories. Note that no special interpretation of the prefix argument is performed. It is just inserted in front of the default object filename. The Prefix typically is a pathname. The -o and -O cannot be used at the same time.

Example:

```
C-6811 test.c -O\upper (output file on \upper\test.r07)
```

-b Make object a library module

By default the compiler generates an object module with the 'program' attribute (see assembler chapter). With the -b switch, the object modules will get the 'library' attribute.

-P**Generate PROMable code**

If a C program contains statically initialized data or depends on static data without explicit initializers set to zero, it cannot be put into read-only storage directly since variable objects must be mapped to RAM which needs to be initialized at power-up time. With the -P command line switch static initializers will be put into its own memory segment which will be mapped into ROM address space and at start-up copied into RAM by the C run-time code. If the recommended linking scheme is used, all operations and memory allocations will be automatically and transparently performed.

-g{OA}**Enable global typecheck**

With the -g switch the compiler enters a mode which is more strict about the use of certain C constructs like:

- Calling undeclared functions
- Undeclared K&R formal parameters
- Missing "return" values in non-void functions
- Unreferenced local or formal parameters
- Unreferenced "goto" labels
- Unreachable code
- "string" or 'c' containing non-printable characters
- Unmatching or varying parameters to K&R functions
- #undef on unknown symbols
- Valid but ambiguous initializers
- Constant array indexing out of range
- Old-Style K&R-functions [_gA only]

In addition to the checks performed at compile-time the -g switch enables the generation of linker type information which is used for module interface checking. The -g command does not generate more code but takes additional time to process in the compiler and in XLINK.

The -g switch can optionally be succeeded by the letter O and/or A. O implies that no object code information is to be generated but full type-check should still be performed within the module. The A modifier forces the compiler to complain about the pre-ANSI type of function declarators that are in this document referred to as K&R.

Note that modules compiled without the -g option are considered as typeless and global declarations in such modules will match (in

XLINK) any declarations in other modules regardless if these modules were compiled with or without the -g option.

-w**Disable warnings**

If the -w flag is given in the command line, warning messages will not be displayed although they will still be counted and will show up in compilation statistics.

-mx**Selects memory model**

See section 3 for more details.

-d**Forces 'static' allocation of 'auto' variables**

This option forces the compiler to allocate auto variables in static memory locations instead of the stack.

-2**Enables double precision math**

This option enables all double and long double types into eight bytes, 64-bit IEEE format, referred to as double precision.

Selection of this option insures that the appropriate library modules are linked in (CL6811?D.R07) to allow the conversion of the double and long double data objects to be treated as 64-bit objects and not the default 32-bit size.

Information on the range of numeric values supported using floating point operations is provided in the float.h file. Absolute values below the smallest limit will be regarded as zero, while overflow conditions give undefined results.

-S	Optimize for speed
-Z	Optimize for size

By default the compiler performs code-optimizations. However, the level and goal of the optimization can be further controlled by the -s and -z switches. Only one of these flags can be activated during a single compilation.

-r{0 1 2}{i}{n}	Generate Debug Information
------------------------	-----------------------------------

Generate debug information. With the -r switch, the compiler will output additional information in the object code which can be used by *debuggers.

The 0, 1, and 2 modifiers are implemented to support different debugger hardware. The source-level *debuggers manual should specify the appropriate modifier. However, for assembly *debuggers, the 0 modifier is sufficient. Note that the compiler will insert one or more NOPs in line with your code after each C line if the 1 or 2 modifier is used. NOPs are added to allow some *debuggers to align C lines with the corresponding assembly code. CAUTION: if NOPs are inserted in your code, the timing of the simulation will not be accurate.

The n modifier suppresses the generation of C source lines in the object file. Most debugging tools use the source file (.C) or a listing file to provide the source code information. The Archimedes symbolic debugging format AUBROF/DEBUG reads the source code lines from the symbolic compiler output file.

The i modifier includes the #include files information in the object code file. A side-effect of the i modifier is that source line records will contain the global (=total) line count, which can affect source line display in some *debuggers.

NOTES

<p><i>For more information on emulators, refer to the EMULATOR.DOC file. For most *debuggers that do not include specific information on how to use Archimedes C compilers, -m should be specified.</i></p>

The `-r` option causes the compiler to include global typechecking information in the object file whether the `-g` (add typecheck information to object file) compiler option is enabled or not.

The `-r` option requires the compiler to use more host memory, reducing the size of modules which can be compiled. Using the `-rn` combination reduces this effect somewhat.

The 1 and 2 modifiers add NOPs to the generated executable code as required by certain debugging tools. Use these modifiers only if your development tools specifically require you to do so.

Global optimization activated by the `-z` and `-s` options may invalidate the source line information (due to rearrangements of the executable code performed by the compiler) which may affect source level debugging.

-y Put "strings" into variable section

By default C string literals are assumed to be read-only. With the `-y` switch activated strings will be generated as initialized variables. Arrays initialized with strings (i.e. `char c[] = "string"`) however, are always treated as ordinary initialized variables.

-c Make plain 'char' = 'signed char'

Makes all plain character data types signed by default (for compatibility with some other C-compilers).

-e Enable target dependent extension

See section 12 for more details.

-l file Generate a list on: `<file> <.lst>` **-Lprefix** Generate a list on: `<prefix> <source> <.lst>`

Use one of these commands to generate a C list file. Filenames are treated analogous to the `-o` and `-O` switches.

-tn Set tab spacing between 2 and 9 (default 8)

In C list files, tabulators are converted to space with the value specified with the -t switch.

-x{DFT2} Generate cross-reference list

When the -x switch is given a list of all global symbols and their meaning will be printed after the C source code. By default the -x switch without any arguments will show all variable objects and all referenced functions, #defines, enums and typedefs. If the -x switch is succeeded by one or more of the arguments D,F,T and 2, the list will be expanded to also include:

- D: Unreferenced #define symbols
- F: Unreferenced function declarations
- T: Unreferenced enum constants and typedefs
- 2: Dual line spacing between symbol entries

-q Put mnemonics in the list

Merges C source code with generated native code in symbolic assembly format on the list file. For debugging purposes.

-T List 'active' lines only (#if etc. true)**-i List #included files**

-pnn Page listing with 'nn' lines/page (10-150)**-F Generate formfeed after each listed function**

-a file **Generate ASM on: <file> <.s07>**
-Aprefix **Generate ASM on: <prefix> <source> <.s07>**

When one of these switches is activated the compiler will generate symbolic assembly code on either a specified file or use the prefix argument combined with the basename of the source file as the target file. The generated code can then be assembled by the Archimedes assembler. <.s07> is an 68HC11 specific extension.

-Hname **Set object module header = 'name'**

Occasionally it is impractical to have the same name on the object module as the base-name of the source file. A typical example is when the compiler is driven by some other pre-compiler that typically always uses the same output file. XLINK would then complain about 'duplicate modules' if several modules were linked. With the -H switch, the default name can be overridden.

-Rname **Set code segment = 'name'**

Executable code is usually put on a segment named CODE. In the cases where a module must be loaded at a very special address, the -R switch makes it possible to generate a unique segment name which can be placed at any address by XLINK. This is useful in bank-switched systems or in systems using multiple separate PROMs as it facilitates setting up different segments in the linker file.

-Iprefix **Add #include search prefix**

To force the include command to search at other directories for a specified file, any number of -I options with directory information can be issued.

Example:

`-I\usr\proj\`

Note that the prefix parameter is only added to the include file name without any interpretation (i.e. -I\usr\proj would make the compiler try to open \usr\projinclude_file rather than \usr\proj\include_file). Also see section 1.18 about Include Files.

-f file Extend command line with <file> <.xcl>

The command line of the C-compiler can be extended to almost any length by using a command file which should contain command line parameters formatted in the same fashion as the ordinary command line. Note that in command files, the newline (<CR>) character is considered equivalent to tabs and spaces (i.e end of file is the end of the command file) which makes command file data more readable on printers etc. The default file extension on command files is .xcl.

-DSYMB Equivalent to: #define SYMB 1
-DSYMB=xx Equivalent to: #define SYMB xx

With the -D switch you can do a command line #define of a symbol. This is particularly suited for configuration purposes or in conjunction with conditional compilation to disable or enable calls to user-written trace routines. Any number of symbols can be defined although you may also have to use the -f switch to get the command line length required. To make it possible to #define string literals etc. double quotes may be used around space separated items to make them appear as one:

```
"-DEXPR=F + g"      #define EXPR F + g
"-DSTRING=""micro proc""      #define STRING "micro proc"
```

-USYMB Equivalent to: #undef SYMB

When the compiler is invoked, a number of pre-defined #define symbols exist. If any of those symbols is in conflict with (i.e., already used) a user #define symbol, that symbol initial definition can be turned off with the -U switch. The pre-defined symbols are:

Archimedes_C	
<u>FILE</u>	"current source filename"
<u>LINE</u>	"current source line number"
<u>TIME</u>	"hh:mm:ss"
<u>DATE</u>	"Mmm dd yyyy"

-G**Open standard input as source**

With the -G option switch, a C program can be read directly from the keyboard. The source (dummy) filename is set to stdin.c.

-S**Silent operation of compiler**

Makes compiler turn off sign-on message and compilation statistics report.

-C**Enable nested comments**

When -C is activated, comments can be nested to any level. Without the -C command line option, the compiler will warn about, but ignore, nested comments. The purpose of the -C flag is to make it easy to "comment out" parts of a C program that could also contain comments.

-K**Enable C++ style comments**

When -K is activated, the compiler will recognize C++ type comments, such as `//`.

Note that the extended command-line data for setting switches can be given through a target-dependent environment variable:

```
set QCC_6811=-V -s -DCONFIG=43
```


16. Include Files

Include files are typically used to keep common definitions available to the different modules forming a program. The include files may be specified in two ways: Within angle brackets (`<file>`) or as a C string literal ("`file`").

When an `#include` directive is found the compiler tries to open the specified file in the following order:

1. Only for "`file`": File prefixed with the directory part of the invoking source file. That is, if the invoking file name is `..\src\sieve.c` the include file "`std.h`" will force the compiler to try to open the file `..\src\std.h`.
2. File name prefixed with the arguments of any optional `-I` switch commands given.
3. File name prefixed with arguments of an optionally defined environment variable named `C_INCLUDE`. Note that multiple search paths can be specified in `C_INCLUDE` by separating arguments with semicolon like:

```
set C_INCLUDE=\usr\proj\;\headers\
```

4. Lastly the file is opened using the 'naked' file name.

Note that arguments to `-I` switch and/or to `C_INCLUDE` are just added in front of the `#include` file name without any interpretation. That is, a directory prefix must also contain the terminating backslash.

If the compiler fails to open a file using the steps above the compiler aborts the current compilation.

17. Compiler Diagnostics

The error messages produced by the C-compiler falls into six categories:

1. Command line errors
2. Compilation warning messages
3. Compilation error messages
4. Compilation fatal error messages
5. Memory overflow message
6. Compiler internal errors

Command line errors

Command line errors occur when the compiler is invoked with bad parameters. The most common situation is that a file cannot be opened. The command line interpreter is very strict about duplicate, misspelled or missing command line switches. However, it produces error messages that point out the problem in detail.

Compilation warning messages

Compilation warning messages are produced when the compiler finds constructs which are typically due to a programming error or omission. Appendix B lists all warning messages.

Compilation error messages

Compilation error messages are produced when the compiler finds constructs which clearly violate the C language rules. Note that the Archimedes C-compiler is more strict on compatibility issues than many other C-compilers. In particular, pointers and integers are considered as incompatible when not explicitly casted. Appendix B lists all error messages.

Compilation fatal errors

Compilation fatal error messages are produced when the compiler finds a user error so severe that further processing is not meaningful. Compilation is terminated immediately after the diagnostic messages are issued. Appendix B lists all compilation error messages (some marked as fatal).

Memory overflow message

The Archimedes C-compiler is a memory-based compiler which in case of a small host system memory or very large source files may run out of memory. This is recognized by a special message:

```
* * * C O M P I L E R   O U T   O F   M E M O R Y * * *
```

```
Dynamic memory used: nnnnnn bytes
```

If such a situation occurs, the cure is either to add system memory or to split source files into smaller modules. However, with 570K RAM the compiler capacity is sufficient for all reasonably sized source files. If memory is on the limit, please note that the -q, -x, -r, and -P command line switches cause the compiler to use more memory.

Another solution is to add extended memory to your system. The presence of extended memory allows you to use the extended memory versions of the compiler, linker, and librarian.

Compiler internal errors

During compilation a number of internal consistency checks are performed. If any of these checks fail, the compiler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to Archimedes technical support group.

18. C-68HC11 Compatibility.

The Archimedes C-6811 implements the full ANSI standard C language and the most important library functions for microcontroller development. This chapter gives a short description of the most significant differences of the proposed ANSI C standard versus the K&R C definition. (Also, review Appendix F: Libraries for an overview of the library functions).

18.1 New ANSI C keywords

In addition to the traditional C language keywords, the C-6811 compiler implements the following ANSI C keywords and constructs:

const

An attribute that tells that a declared object is non-modifiable:

```
const int i;      /* constant int */
const int *ip;    /* variable pointer to constant int */
int *const ip;    /* constant pointer to variable int */

const int i;      /* constant int */
const int *ip;    /* variable pointer to constant int */
int *const ip;    /* constant pointer to variable int */

typedef struct
{
    char *command;
    void (*function)(void);
} cmd_entry;

const cmd_entry table[] =
{
    "help",      do_help,
    "reset",     do_reset,
    "quit",      do_quit
};
```

volatile

An attribute with same syntax as const but tells that the value in the object may be modified by hardware and should not be "optimized out".

signed

Analogous to unsigned. Can be used before any integral type-specifier.

void

Is a type-specifier that can be used to declare function return values, function parameters and "generic" pointers.

```
void f() /* A function without return value */
type_spec f(void); /* Function with no parameters */
void *p; /*Generic pointer.Can be casted to any other pointer and
is assignment compatible with any pointer type */
```

enum

Enumeration constants is a way to generate sequential integer constants that can replace #defines in some contexts.

```
enum {zero,one,two,step=6,seven,eight};
```

Data types

The complete set of basic data types now includes:

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
* /* Pointer */
```

Observe that pointers are not integers, i.e. the C-compiler is very strict in typechecking. This is especially crucial to look out for if you are porting 'generic C-code' which you have used with an older type of a compiler, which may have allowed integers to be freely assigned to pointers (and vice-versa) without complaining. If you must assign a value to a pointer variable, "cast" the integer to a pointer type, e.g.

```
char *ptr;
int i;
ptr = (int *)i;.
```

Function prototypes

Function prototyping is a way to specify parameters in function declarations as well as in function definitions that has been added by ANSI. Prototyping results in safer and under some circumstances more compact programs. The following example shows the difference between K&R declarations and prototypes:

K&R	Prototype
<pre>extern int func(); abort(s) char *s; { }</pre>	<pre>extern int func(long val); void abort(char *s) { }</pre>

Now if "func" is called with an "int" parameter the K&R version will fail to work if "int" has a different size than "long" while the prototyped version will extend the "int" argument in the same way as an assignment operator would. When prototyped functions are used, arguments are always checked for compatibility with the declaration, whereas K&R type of declarations are only checked when the -g option is on. The prototyped format also specifies how to denote a variable number of arguments. This is done by writing three dots in the place of a formal parameter. Also see library function "printf".

If external or forward references to prototyped functions are used, a prototype declaration should appear before the call.

Hexadecimal numbers in strings

In addition to the original octal constants (\nnn), ANSI allows hexadecimal constants (one to three digits) by using a backslash followed by x and the hexadecimal digits.

```
#define Escape_C "\x1b0C"
```

Note that the zero was needed to avoid that the letter "C" was interpreted as a part of the hexa decimal constant.

Structure and union assignments

The current ANSI draft allows functions and the assignment operator to have arguments that are "struct" or "union" type rather than only pointers to such objects. Functions may also return structures or unions.

```
struct s a,b; /* struct s declared earlier */
struct s f(struct s parm);
a = f(b);
```

18.2 Additional ANSI Adaptations

1. Unrecognized #pragmas are not flagged as errors. They are now diagnosed as a "#pragma error" message having the same format as an ordinary error message. However, these #pragma errors are treated as warnings, i.e, they can be ignored as long as the -g option is not used. If, however, the -g option is used, these errors are treated as regular errors which disable code generation and the creation of an object file.
2. Incomplete structure declarations generate warning [38] "Tag identifier NAME was never defined" when the -g option is used.

Example:

```
struct NAME *ptr;
```

3. It is not illegal to declare arrays or structs with "register" storage.
4. Static forward function declarations no longer generate error messages if these functions were never defined or referred to.

18.3 Additional Language Extensions

1. The last member of a struct or union definition may be an array with no size or size set to zero. The size of the struct will then be that of the other members.

Example:

```
typedef struct
{
    char    mess_nr;
    int     p1;
    int     p2;
    char    mess_arg[0];
} message_header;
```

This declaration could then be used as:

```
mess_ptr = malloc(sizeof(message_header) + nr_of_bytes);
```

2. Linkage Model

There are some differences between C compilers currently on the market when variable objects are to be shared among modules (= files). For functions most C compilers perform identically. In Archimedes portable C compiler, the scheme recommended in the ANSI supplementary document Rationale For C is used. This linkage model is called "Strict REF/DEF" and requires that all modules except one use the keyword `extern` before the variable declaration.

Example:

Module #1	Module #2	Module #3
<code>int i;</code>	<code>extern int i;</code>	<code>extern int i;</code>
<code>int j=4;</code>	<code>extern int j;</code>	<code>extern int j;</code>

18.4 Pre-processor Directives

In addition to the traditional C pre-processor directives, the C-6811 compiler implements the following directives:

#elif expression

```
#elif <expr>  
  C lines
```

is equivalent to:

```
#else  
#if <expr>  
  C lines  
#endif
```

#error any_valid_C_tokens

This directive is supposed to be used in conjunction with conditional compilation. When the **#error** directive is found the compiler stops with an error message (if compilation was not turned off by earlier **#ifdef**'s etc).

#message any_string

New **#message** pre-processor directive was added to help checking whether or not a conditional compilation occurred as expected.

Example:

```
#ifdef FRENCH  
#message "French version of the XYZ electronics home robot"  
printf("Bonjour le monde, ca va?\n");  
#endif  
#ifdef USA  
#message "US version of the XYZ electronics home robot"  
printf("Hello world, we will soon send the marines!\n");  
#endif
```

19. C-Compiler Extensions

To better support generation of target-dependent code while still using a single source file for more than one target, a number of small C language extensions has been added to the Archimedes C-Compilers. For applications, study the #include file stdarg.h.

1. The macro `__TID__` returns an integer constant with the following coding:

15	8 7	4 3	0
Target_IDENT	-v option	-m option	

Target_IDENT: A unique number for each target. (execute `printf("%d", __TID__ > 8)` to get the value)

- v option: If compiler has a -v option this field holds the value (otherwise is 0)
- m option: If compiler has a -m option this field holds the value (otherwise is 0)

2. The unary operator `_argt$` has the same syntax and argument as `sizeof` but returns a normalized value describing the type of the argument according to the following:

unsigned char	1
char	2
unsigned short	3
short	4
unsigned int	5
int	6
unsigned long	7
long	8
float	9
double	10
long double	11
pointer/address	12
union	13
struct	14

3. The reserved word `_args$` returns a char array (char *) that contains a list of elements describing the formal parameters of the currently compiled function (i.e. `_args$` can only be referred to inside function definitions). Parameter list layout:

-->

NT 1	SP 1		NT n	SP n	\0
------	------	--	------	------	----

Description

NT holds a Normalized Type (same values as for `_argt$`) whereas SP holds the Size of a Parameter (although parameters in excess of 127 bytes will only show up as 127 on the list).

The list ends when a type field with a zero value has been found (no more parameters or a "..." declaration).

4. The character \$ has been added to the set of valid characters in identifiers to maintain compatibility with DEC/VMS C.
5. The ANSI-specified restriction that the `sizeof` operator cannot be used in `#if` and `#elif` expressions has been eliminated.

20. Sample Code

The short program below illustrates the use of the Archimedes C-6811 cross-compiler with the `-L` and `-q` compiler switches initialized. The `-L` switch generates a list file. The `-q` switch puts mnemonics in the list, i.e. merges C source code with generated native code in symbolic assembly format in the list file, which is useful for debugging purposes.

Source [code.c]:

```

/* code.c: Example on mixing C and assembly code in a listing file */

char c[] = "Hello World";
int i=78;
int j;
void main(void)
{
    static int a = 4;
    float ss;
    i += j;
}

```

Invocation:

C-6811 code -L -q

```

#####
#
#Archimedes 68HC11 C-Compiler V3.30B/DXT      16/Apr/92  10:29:42#
#
#      Memory model = large
#      Source file   = code.c
#      List file     = code.lst
#      Object file   = code.r07
#      Command line  = code -L -q
#
#                                     (c) Copyright Archimedes 1992 #
#####

```

```

\ 0000      NAME    code(16)
\ 0000      RSEG    CODE(0)
\ 0000      RSEG    DATA(0)
\ 0000      PUBLIC  c
\ 0000      PUBLIC  i
\ 0000      PUBLIC  j
\ 0000      PUBLIC  main
\ 0000      EXTERN  ?CL6811_3_20_L07
\ 0000      RSEG    CODE

1          /* code.c: Example on mixing C and assembly code in a
listing file */
\ 0000      P68H11
2
3          char c[] = "Hello World";
4          int i=78;
5          int j;
6          void main(void)
7          {
\ 0000      main:
\ 0000 3C      PSHX
\ 0001 3C      PSHX
8          static int a = 4;

```

```

      9      float ss;
     10      i += j;
\      0002 FC000E      LDD      j
\      0005 F3000C      ADDD     i
\      0008 FD000C      STD      i
     11      }
\      000B 38          PULX
\      000C 38          PULX
\      000D 39          RTS
\      0000            RSEG     DATA
\      0000            c:
\      0000 48656C6C      FCC     'Hello World'
\      0004 6F20576F
\      0008 726C64
\      000B 00          FCB      0
\      000C            i:
\      000C 004E          FDB      78
\      000E            j:
\      000E 0000          FCB      0,0
\      0010            ?0000:
\      0010 0004          FDB      4
\      0012            END

```

Errors: none

Warnings: none

Code size: 14

Constant size: 0

Static variable size: Data(18) Zpage(0)

The Time Saver



TM

ARCHIMEDES
SOFTWARE

Assembler

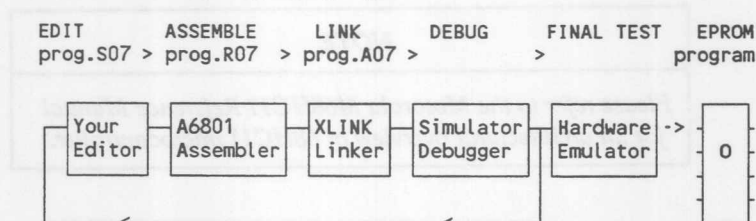
ASSEMBLER

1. Assembler Overview

The Archimedes Software A6801 assembler is a powerful relocating, macro cross-assembler for the 6801 family of microcontroller chips (6801, 6301, and 68HC11). A6801 is used in conjunction with the Archimedes XLINK Linker and XLIB Librarian (included in the standard Assembler Kit). Together with your other favorite development tools, the A6801 Assembler Kit helps you efficiently develop software for 68HC11-based embedded controller applications.

The A6801 Assembler is available for the IBM PC/AT or compatible (MS-DOS), DEC (VMS or Ultrix), HP-3000 (Unix), and Sun (Unix) host environments. It assembles 68HC11 assembly language source files into relocatable object modules, which are linked together using the Archimedes XLINK linker. The resulting output file, formatted in Motorola S-Record, or in one of over 20 other supported formats, can then be downloaded to a hardware emulator or PROM programmer for testing.

The diagram below illustrates the flow of a typical development cycle for A6801 assembly language programs



The A6801 Assembler Kit may be used as a standalone development product or in conjunction with the Archimedes C-6811 Cross-Compiler. C-6811 is a full ANSI-C compiler with a rich set of extensions to support microcontroller programming. This combination allows you to code the "high-level" portions of your programs in C and the time-critical sections in assembler -- giving you the best of both worlds.

1.2 Characteristics and Features

1.2.1 Basic Features

The A6801 Assembler offers a number of powerful features for developing embedded controller software:

- Support for all standard 68HC11 chip features
- Compatible with most other assemblers (see the "Compatibility" section in this chapter)
- Support for modularized programming with multiple modules per source file
- Up to 255 significant (upper/lower case) characters in global symbols
- Virtually unlimited number of symbols (host memory limited)
- Support for link-time typechecking
- Includes a powerful macro-processing language
- Full set of directives for list-file control
- 32-bit structure for all internal calculations
- A two-pass assembler which executes as a single program -- no temporary files are created during execution
- Easy interface to Archimedes C-6811 programs
- Symbolic debugging support with most hardware emulators

1.2.2 68HC11 Chip Support

NOTE

Please refer to the Motorola M68HC11 Reference Manual for an architectural overview of 68HC11 microcontroller.

The major differences between the proliferation chips lies in the variety of on-chip peripherals, memory size or other enhancements that are included.

1.2.3 Compatibility

The A6801 Assembler is highly compatible with other 68HC11 Cross-Assemblers. In general, it is compatible in the following areas:

- 68HC11 instructions (mnemonics and syntax)
- Constant defining directives (FCB, FDB...)
- Space allocation directives (RMB,...)
- Value directives (EQU, SET,...)
- Location directives (ORG)
- Delimiters (space, tab, semicolon,...)
- Constants ('ABC', 2Bh,...)
- Labels
- Basic operators (+, -, *,...)
- Absolute output format (created by XLINK)

A6801 is not compatible (or partially so) with the other assemblers in the following areas:

- Extended operators
- Assembler options and controls
- Relocation directives (RSEG, ASEG,...) are handled differently
- Conditional assembly directives
- Macro processing

Many of the differences listed above only have a different syntax from that employed in other assemblers, so that the same functions can be readily achieved in A6801 programs with minor changes.

Also note that the relocatable object file format generated by A6801 (See Appendix J, Glossary:UBROF), and that used by all other assemblers for their relocatable object files, are not the same. Therefore the Archimedes XLINK Linker must be used to link A6801 programs. XLINK cannot be used to link files assembled with any other assembler.

IMPORTANT

Relocatable object files created with the A6801 assembler (or the C-6811 compiler) CANNOT be linked with other vendors' files as no common format exists for relocatable code.

2. Using the Assembler

2.1 Installation

Before proceeding with this section, you must install the A6801 software on your computer system. This involves creating directories on your hard disk and copying the necessary files from diskette onto the system. Appendix-A contains installation instructions and lists the hardware requirements for your system.

IMPORTANT

If you have not already done so, please install the A6801 software at this time on your computer as directed in Appendix-A: Installation.

If you wish, you can now verify that the software is correctly installed on your system by running an MS-DOS batch file named ATEST.BAT. This batch file assembles a set of files that contain all instructions and operations accepted by the assembler.

Change directories to where you installed the Assembler software (e.g., C:\A68) and run the ATEST BAT file:

```
C:\A68> ATEST <return>
```

This batch file takes a 68HC11 source file named A6801.S07 and assembles it. If the Linker terminates normally, meaning that the assembly process was successful, the batch file displays a message indicating so.

2.2 File Naming Conventions

The A6801 Assembler (and XLINK Linker) use the following naming conventions for the various files they process:

Filetype	Description
.S07	Source files read by the Assembler (default); this is also the default filetype for "include" files
.MSA	Optional source file default filetype
.INC	Optional filetype for "include" source files
.R07	Relocatable output files created by the Assembler
.LST	Listing files created by the Assembler (default)
.A07	Absolute, linked code in symbolic format; created by the XLINK Linker
.HEX	Absolute, linked code in Intel-Standard "HEX" format; created by the XLINK Linker
.XCL	XLINK Linker command files (default)
.MAP	Map/cross-reference files created by XLINK

2.3 Editing Source Files

Source files for the A6801 assembler can be created with any text or program editor that produces ASCII files, such as PI-Edit (Iliad Group), Brief (Solution Systems), The Norton Editor (Peter Norton) or, for the truly adventurous, EDLIN. If you choose to use a general purpose word-processing program, such as Word or Word Perfect, to write 68HC11 programs, be sure to use the "non-document" or ASCII-file mode of operation when you save files to disk.

For simplicity, we recommend that you use the default filetype of ".S07" for 68HC11 source files. Include files should also be given a filetype of ".S07".

In the next section, Creating Assembler Programs, we will examine the actual structure and syntax for 68HC11 source programs.

2.4 Basic Operation

In this section we will get acquainted with the assembler operations by assembling and linking a small demo program.

2.4.1 Overview

The A6801 assembler can be executed in one of two ways:

1. In command line mode, where all parameters for the assembly are entered on the command line. For most programmers this will be the most frequently-used mode of operation as it lends itself to batch or make-file operation.
2. In prompted mode, where you enter only "A6801", press <Enter>, and the Assembler prompts you for all file name(s) and options. This mode is for occasional use where you might not remember the order of parameters or the option list.

2.4.2 Simple Command line Examples

The most basic command line for running the A6801 assembler is shown in the example below, which assembles and links the sample program ADEMO.S07, which is included on the distribution diskettes (user input is shown underlined):

```
c> A6801 ADEMO
```

```
Archimedes 6801 Assembler V2.00/DOS  
(c) Copyright Archimedes 1991
```

```
Errors:  None      #####  
Bytes:   17        # ADEMO # <--- Module name  
CRC:     8F6F      #####
```

This command line runs the Assembler and causes it to look for a source file named ADEMO.S07, since the default filetype for source files is ".S07". It then assembles the program and creates an output

file named ADEMO.R07, which contains relocatable code (in UBROF format, see Glossary in Appendix J) ready to be linked with XLINK. In this example, no listing file will be created.

Please note that the module name can be different from the file name. Module names are specified within the source file itself, and a single source file can contain multiple modules. (see Glossary: Modules in Appendix J). The concept of modules and multi-module programming will be discussed in greater detail in the next section.

To have the assembler generate a listing you must specify a name for the list device (or disk file) as a second parameter on the command line, as shown below:

```
C> A6801 ADEMO LPT1
```

This command line assembles the program ADEMO.S07 and prints a listing on a printer attached to the MS-DOS LPT1: printer port. If you want to create a listing in a disk file instead, substitute a filename for "LPT1". For example, the next example creates a list file named ADEMO.LST (the default filetype for list files is ".LST"):

```
C> A6801 ADEMO ADEMO
```

The format for list files is explained in this section under Assembler Listings.

Now that we have assembled a simple source file, we must use XLINK to link it into an absolute, 68HC11-executable program.

2.4.3 Linking the Demo Program

The relocatable object code for our sample program is now contained in a file named ADEMO.R07. To use XLINK to link the example program, enter the following command line:

```
C> XLINK ADEMO -c68HC11 -ZCODE=C000 -ZUDATA=0 -o ADEMO.A07
```

```
Archimedes Universal Linker V4.38/DXT  
(c) Copyright Archimedes 1991
```

```
Errors: none  
Warnings: none
```


NOTE

Upper/lower case IS significant for XLINK command line options.

The XLINK command line above links the ADEMO.R07 relocatable file to produce an absolute program file that can be downloaded and executed on your 68HC11 target hardware.

The "-c68HC11" option identifies the CPU type; the "-ZCODE=C000" locates the program's only code segment at location C000; the "-ZUDATA=0" locates a data segment at 0H; and "-o ADEMO.A07" determines the output file name. The default output file format is used, Motorola S-Record.

To examine the Motorola output file, enter:

```
C> TYPE ADEMO.A07
```

```
S00B000043535441525455507E
S10F60008E2200BD6020BD600C7E601D7F
S105FFFE60009D
S106601D7E601D81
S1136020CE2001CC2000BD6076CE60AE3CCE60AE0A
S11360303C30EC00A3022713EE02EC00EE02BD603C
S11360407630CC0004E302ED0220E7CC60AEED0232
S1136050CC60AEED00CE20003CBD609030CC60AE94
S1136060ED04CC60AEED02CC2001ED00BD609031BA
S1136070313131313937363C30EC00A302270B52
S1136080EE026F00088F30ED0220EF3131313139EB
S113609030EC06A3042716EE04A60036088F30ED74
S11160A00532EE03A700088F30ED0220E4392C
S113600C5FF72000F62000C10A24057C200020F450
S104601C3946
S90360009C
```

This file could be downloaded to a PROM programmers, or a hardware emulator.

The XLINK Linker has many other command line options and parameters. See 2.8 Linking A6801 Programs as well as the XLINK section of this manual for more detailed information.

2.5 Command line Operation

This section covers Assembler command line syntax and options.

2.5.1 Command line Syntax

The entire A6801 Assembler program is contained in a single executable file named A6801.EXE. No overlays are needed and no temporary files are created during the assembly process.

The complete syntax for A6801 Assembler command lines is described below. A number of examples can be found in 2.5.3 Command Line Examples.

A6801 command line syntax:

A6801 [<Source File>], [<List File>], [<Object File>], [<Options>],...

Options:

X	put cross-reference and symbol tables in listing
P=nn	page the listing with "nn" lines/page
E	include only lines in the listing with errors
F	format the listing into fixed fields
W	enable the "wide" format for list output
S	put local symbols in the output object file
?	display a list of available assembler options

Note the following:

- Elements shown in square brackets ([]) are optional
- Either a comma (,) or a space may be used as a delimiter between elements
- Filenames and options may be entered in either upper or lower case (not significant)
- To specify an empty or a missing parameter, use double comma delimiters (,,)

Each of these parameters and options is explained in the next section.

2.5.2 Command line Parameters and Options

NOTE

A standard MS-DOS drive and pathname designator (d:path_name) may also be used when specifying any filename on the Assembler command line.

Source File

This is the name of the assembly-language source file to be assembled. Two default filetypes are recognized by the Assembler: ".MSA" and ".S07". If a source file of type .MSA cannot be located in the current working directory (or in the directory specified in the path for the source file), a filetype .S07 will be assumed. The preferred filetype for 68HC11 Assembler source files is .S07.

If a "Source File" is not specified on the command line, or if the specified file cannot be found, the Assembler will stop and prompt you for the name of the source file, list file, object file and options; see 2.6 Prompted Operation, below.

List File_{opt}

This is the name of a disk file or an MS-DOS device that the Assembler is to use for its list output. Use the device name "PRN" or "LPTn" (where n is 1, 2 or 3) to direct the output to your printer. Use the device name "CON" to have the listing appear on your screen.

Specify a valid filename and optional filetype to place the listing in a disk file. If a list file is specified without a filetype, the Assembler will add a type of ".LST" to the file name.

If this parameter is left off of the command line, a listing will NOT be created.

See 2.9 Assembler Listings later in this chapter for an example of a list file.

A number of Assembler Directives can be placed in the .S07 source file that will affect the contents and format of the listing. For example, by using the "LSTOUT +" and "LSTOUT -" directives it is possible to control which sections of the code will appear in the listing. For more information on these directives, see the Assembler Directive Reference section of this manual. Also see the list of command line Options, below.

Object File_{opt}

This parameter is used to specify the name of the relocatable output file created by the Assembler. If this name is not specified, the Assembler will create an output file with a filetype of .R07, with the primary name the same as the source file name.

The object files created by the A6801 Assembler are in an Archimedes-proprietary format known as UBROF (Universal Binary Relocatable Object Format, see Glossary: UBROF in Appendix J). The .R07 object files created by the Assembler must be linked with the XLINK linker to produce absolute, 68HC11-executable programs.

If you do not wish to create an object file (e.g., you only want to see if your program assembles correctly), specify the "NULL" device name as the object file.

Options (may be entered in upper or lower case):

X

Causes a cross reference and symbol table to be added to the listing file. The default condition of this option is that these tables will NOT be included in the list file. See 2.9 Assembler Listings below for an example of cross reference and symbol tables. This option has the same effect as the "LSTXRF" assembler directive (see 5.9.16 LSTXRF).

P=nn

Specifies a page size (lines/page) for the listing where "nn" is a decimal number between 10 and 150, inclusive. The default condition if this option is not specified is a continuous (non-paged) listing. This option has the same effect as the "LSTPAG +" with the "PAGSIZ <nn>" directive (see 5.9.9 LSTPAG and 5.9.10 PAGSIZ).

E

Include only lines in the listing that contained assembly errors. This option could be useful when assembling large programs for the first time, as it can save you a lot of paper and/or disk space. The default condition is to include all error and non-error lines.

F

Formats the list file into fixed-position fields regardless of how the input source file is formatted. The default if this option is not specified is that the input source appears in the listing "as is" with no formatting (except that tabs are expanded).

W

Causes the Assembler to format the list output using a special "wide" format that identifies the current nesting level and location in a source file where multiple levels of "include" files are in use. This option is equivalent to the "LSTWID +" assembler directive (see 5.9.7 LSTWID). The wide listing format is described in 2.9 Assembler Listings.

S

This option causes the Assembler to put ALL symbols in the relocatable object file, including local symbols. Without this option, only those symbols that have been declared as PUBLIC will be passed on to the XLINK Linker to appear in the final symbol table. This option is the equivalent of the "LOCSYM +" assembler directive (see 5.3.4 LOCSYM).

IMPORTANT

You must use the PUBLIC directive to make symbols available to debuggers, simulators, and emulators.

?

This causes the Assembler to display a menu of command line options and prompt you to enter one or more before proceeding with the assembly. This feature is useful when you have forgotten the letter of the desired option (useful for those 3:00 AM programming sessions). Also see 2.6 Prompted Operation, for an example. ? (Assembler option)

2.5.3 Command line Examples

The following A6801 command line examples will illustrate the use of the parameters described above:

```
A6801 FOOBAR,PRN,,X,P=66
```

assembles FOOBAR.S07, sends the listing to the MS-DOS PRN printer device, puts the cross-reference and symbol tables in the listing and enables paging with a page length of 66 lines/page.

```
A6801 DOLITTLE,,BIPPLE
```

assembles DOLITTLE.S07, does not produce an output listing, and places the relocatable output code in BIPPLE.R07. The lack of a list-file parameter is indicated by the two commas (,,).

```
A6801 BILGE,BILGE,,W,S,P=60
```

assembles BILGE.S07, puts the listing in BILGE.LST, the relocatable object code in BILGE.R07, enables the "wide" listing option, outputs local symbols, and sets the pagination length to 60 lines/page.

```
A6801 RUMPUS.ASM CON,,X F
```

assembles RUMPUS.ASM and sends the listing to the console, including a cross reference/symbol table. The listing will be formatted into fixed fields (F).

```
A6801 BUGSRFUN,LPT2,OUT,?
```

assembles BUGSRFUN.S07, sends the listing to the printer attached to the LPT2 device, and the relocatable object code to file OUT.R07. Because of the "?" in the options field, the Assembler will prompt with a menu of options before doing the assembly.

```
A6801 FRISBEE \A68\LIST\FRISBEE NULL E
```

assembles FRISBEE.S07, places the listing file in \A68\LIST\FRISBEE.LST, and sends the output code to the "bit bucket" (the MS-DOS "NULL" device). Only lines with errors will be included in the list file.

2.6 Prompted Operation

If you enter "A6801" by itself, the Assembler enters a mode where you will be prompted to enter each command line parameter. This mode of operation is useful for occasional Assembler users who may not remember the command line parameters. The parameters and options you enter in prompted mode are the same as described in the above section for command line operation.

The Assembler session below illustrates the use of "prompted" mode (user input is underlined):

```
C> A6801
```

```
Archimedes 6801 Assembler V2.00/DOS
(c) Copyright Archimedes 1991
```

```
Source file [.msa/.S07]= ADEMO
```

```
List file [.lst]= ADEMO
```

```
Object file [ADEMO.R07]=
```

```
Options= ?
```

```
Options available:
```

```

X   Cross reference and symbol table
P=nn Paged list with <nn> lines/page
E   List only errors
F   Formatted list
W   Wide list with file nest level
S   Put local symbols in object
```

```
Options= XFP=66
```

```

Errors: None          #####
Bytes:  17            # ADEMO #
CRC:    8F6F          #####
```


In the above example, the source file is ADEMO.S07, the listing file is ADEMO.LST and the relocatable object file is ADEMO.R07 (the default filetype is shown in []). Entering a "?" caused the menu of assembler options to be displayed.

As with command line operation, if a list device or file name is not specified, a listing will not be produced.

2.7 Error Messages

All Assembler error messages are issued as complete messages. An example session resulting in an error is shown below:

```
Error in 14: Invalid instruction
jcx      less      to see if its less than 10
      ^
Errors:  1          #####
Bytes:   12         # BAD DEMO 1 #
CRC:     8B91       #####
```

The first line consists of a diagnostic message with the line number in the source file where the error occurred. Then, the erroneous source line itself is displayed with a pointer (^) to the position in the line where the error occurred.

If include files are used (see 3.4.2 Include Files), the diagnostic message will show both the total source lines read to that point as well as the relative line number and name of the file where the error occurred:

```
Error in 296/43 in "GLOBDEF.S07" : Invalid instruction
dv      5      Reserve parameter space
```

The above message indicates that the error occurred at "global" line 296 (the 296th line read by the Assembler), also known as "local" line 43 within the GLOBDEF.S07 include file. To locate this error in your program, load the indicated file in your editor and jump to the indicated "local" line number.

Error messages will always be printed on the screen and will appear in the listing, if one has been specified. Also see the "E" option, described earlier, which causes only erroneous lines to be included in the listing. This is a useful option when first assembling large "unknown" programs.

The Assembler will print as few error messages as possible and report them as soon as possible. In other words, if an error is found and reported in the first pass, it will in most cases not be reported in the second pass.

In addition to Assembler error messages, you may also get Pascal (the implementation language) I/O error messages. These messages are normally easy to interpret.

IMPORTANT!
<i>XLINK will abort the linkage of modules that contain assembly errors!</i>

Please refer to Appendix C: Assembler Error Messages for a complete listing of A6801 error messages.

2.8 Linking A6801 Programs

This section only presents an overview of the Linker operation for the Assembler user. Refer to the Linker section of this manual for more details.

2.8.1 Overview

The Archimedes XLINK Linker is a powerful relocating linker/locator that has many features, options and modes of operation. This section will briefly cover basic Linker operation with A6801 assembly-language programs and present a few examples.

As its input, XLINK reads relocatable object files (.R07 filetype) which must be in the Archimedes-proprietary format known as UBROF (see Appendix J: Glossary: UBROF). These files can be produced by the A6801 Assembler or by the C-6811 C compiler. You cannot use XLINK to link non-Archimedes object files, and other linkers cannot be used to link A6801 or C-6811 relocatable files.

For output, XLINK produces an absolute, 68HC11-executable program file that can be downloaded to most PROM programmers or hardware emulator target boards. A number of symbolic and non-symbolic output formats are supported including the Motorola S-

record format. Refer to section 5.1 Summary of XLINK Output Formats in the Linker chapter for a complete list of formats.

2.8.2 XLINK Command Lines

There are two general ways in which XLINK can be used:

Command line Mode: All input files and options are specified on the XLINK command line:

```
XLINK [-Options...] Input Files... [-Options...]
```

This method is best suited for linking smaller assembler programs with only a few modules and/or segments. Some prefer this method as it gives you direct, command line control over all the linkage options.

Command-File Mode: Input files and options are placed in an XLINK command file (.XCL filetype) and the command file is specified on the XLINK command line using the "-f" option:

```
XLINK -f Command File
```

Large multi-module, multi-segment assembler programs, as well as all C programs, should always be linked using the command-file (.XCL file) method, as entering all the necessary parameters on a single XLINK command line can become impractical, if not impossible. This method also makes it easier to maintain and update programs. (Please refer to the "skeletal" LNK6811?.XCL files included in the compiler package when linking C-6811 programs).

Whichever method you use for linking your A6801 programs, the same set of XLINK commands are used. The two modes of XLINK operation may also be mixed.

NOTE

For our purposes here in this section we are looking primarily at the command line mode of XLINK operation. Refer to section 3.3 XLINK Command Files in the Linker chapter for information on using XLINK command (.XCL) files.

The XLINK parameters and commands most commonly used when linking assembler programs are listed below. See section 4.3 XLINK Command Detail in the Linker chapter, or simply type "XLINK" by itself from the DOS prompt, for a full parameter list:

Input Files

List of one or more relocatable object files (.R07 filetype) to be linked in the listed order; each object file can contain multiple modules (see section 3.2.3 Modules for a discussion of modules)

Options

-c68HC11

define the CPU type as 68HC11-compatible

-Zsegment list=start address

List of segments to link in specified order beginning at start_address (entered in hex); see section 3.2.4 Segments for a discussion of segments

-o file.typ

Put the absolute object code in file.typ (default is AOUT.A07 if a file is not specified)

-l file.MAP

Generate a segment/module "map" file to file.MAP (use PRN for printer or CON for console output)

-x

Include cross-reference tables in the map file

-Fformat

Set the output format type for the output file (default is Motorola S-record format); symbolic and non-symbolic formats are supported -- see Linker chapter, section 5.1, Summary of XLINK Output Formats.

-f Command_File

Extend the command line with Command_File.XCL, which can contain additional XLINK arguments.

2.8.3 Command line Examples

The following examples illustrate the command line usage of XLINK for linking A6801 assembler programs. Refer to the Linker chapter for a full description of XLINK operation.

NOTE
<i>Upper/lower case IS significant for XLINK command line options.</i>

In all of the examples that follow, the "-c68HC11" option is needed to tell the Linker that 68HC11 code is being processed.

```
XLINK SIMPLEX -c68HC11
```

links SIMPLEX.R07 to a default output file named AOUT.A07 in Motorola "Hex" format. The source file should only contain "ASEGs" (absolute code) as no segments are specified on the command line (see section 5.4.2 on ASEG).

```
XLINK FRAMBLE -c68HC11 -ZCODE=0
```

links the file FRAMBLE.S07 and produces an output file with the default name of AOUT.A07 in the default format of Intel-Hex. The segment CODE is linked at address 0 (-Z).

```
XLINK FRAMBLE -c68HC11 -ZCODE=0 -o FRAMBLE.HEX -x -l FRAMBLE.MAP
```

as above, but the output file is named FRAMBLE.HEX (-o option) with a listing file name FRAMBLE.MAP (-l) including cross-reference tables (-x).

```
XLINK GOOSEBIN GOOSEBOX -c68HC11 -ZCODE0=C000 -ZDATA1=0 -o GOOSE.HEX
```

```
XLINK -f BIGPROJ
```

gets XLINK commands from text file BIGPROJ.XCL.

```
XLINK -f BIGPROJ PLUTO -ZXDATA1=8000
```

as above, but also links in file PLUTO.R07 and locates the XDATA1 segment at 8000 (this type of use allows you to "add" XLINK commands to command-file-controlled linkage).

2.9 Assembler Listings

This section explains how to control and interpret Assembler listings.

2.9.1 Overview

The A6801 assembler produces a listing if a List File parameter is specified on the command line as shown in the example below. The listing can be directed to a disk file or to an MS-DOS device: LPT1, LPT2, LPT3, COM1, COM2 or PRN for printers; CON for the console. For example:

```
A6801 LAST LAST
```

assembles LAST.S07 and produces a listing in file LAST.LST, while:

```
A6801 LAST LPT1
```

assembles LAST.S07 and prints a listing on the printer attached to LPT1.

2.9.2 Listing Controls

There is a number of A6801 command line options and Assembler Directives that affect the contents and format of list files. These are summarized below:

Command line Listing Options

- X put cross-reference and symbol tables in listing
- P=nn page the listing with "nn" lines/page
- E include only lines in the listing with errors
- F format the listing into fixed fields

- W enable the "wide" format for list output

Listing Directives

LSTOUT+, LSTOUT-	turn listing on off
LSTCND+, LSTCND-	conditional assembly list on off
LSTCOD+, LSTCOD-	abbreviate initialized code list
LSTMAC+, LSTMAC-	macro definition list on off
LSTEXP+, LSTEXP-	macro expansion list on off
LSTWID+, LSTWID-	include nesting list on off
LSTFOR+, LSTFOR-	reformat listing lines on off
LSTPAG+, LSTPAG-	paginate listing on off
PAGSIZ <expression>	page break after <expression> lines
PAGE	immediate page break command
TITL '<string>'	title pages with <string>
PTITL '<string>'	PAGE plus TITL
STITL '<string>'	subtitle pages with <string>
PSTITL '<string>'	PAGE plus STITL
LSTXRF	generate cross reference table

Refer to section 5.9, Listing Directives, for more information on how these directives are used.

2.9.3 A Sample Listing

The listing from our sample program, ADEMO.S07, is shown below (the list file is called ADEMO.LST, and can be reproduced by running the ADEMO batch file). This listing is for a single-module source file. Multi-module listings are described in the next section.

NOTE

Listings produced by the Assembler show the relative load addresses for relocatable objects within each segment. The Assembler cannot show absolute addresses as these are determined at link time. The cross-reference listing produced by the XLINK Linker can be used to determine the final, absolute load address for all symbols. However, the Linker does not produce a full absolute program listing.

```
#####
#
#Archimedes 6801 Assembler V2.00/DOS [1]    21/Apr/92  15:16:22 [2] #
#
#      Source   =  ademo.s07 [3]          #
#      List     =  ademo.lst              #
#      Object   =  ademo.r07              #
#      Options  =  x f [4]                #
#
#                                           (c) Copyright Archimedes 1991 #
#####
```

```
[5] 1 0000                                NAME  ADEMO
2 0000                                PUBLIC c
3 0000                                PUBLIC main
4 0000                                RSEG   CODE
5 0000                                P68H11
6 0000                                main:
[6] 7 0000 5F                            CLRB
8 0001 F70000 [7]                        STAB   c
9 0004                                ?0001:
10 0004 F60000                            [8] LDAB   c
11 0007 C10A                            CMPB   #10
12 0009 2405                            BCC    ?0000
13 000B                                ?0002:
14 000B 7C0000                            INC    c
15 000E 20F4                            BRA    ?0001
16 0010                                ?0000:
```

```

17 0010 39          RTS
18 0000          RSEG  UDATA
19 0000          c:
20 0000          RMB  1
21 0001          END

```

```

Errors:  None [9]      #####
Bytes:   17  [10]     # ADEMO # [12]
CRC:     1697 [11]    #####

```

Symbol and Cross Reference Table

=====

Symbol	Value	Type	Defline	Refline
--------	-------	------	---------	---------

Segment Definitions

=====

[13] CODE	0011	S00R	4	
UDATA	0001	S01R	18	

External Symbols

=====

[14] Public Symbols

=====

[15] c	0000	R01E	19	2	8	10	14
main	0000	R00E	6	3			

Local Symbols

=====

[16] ?0000	0010	R00	16	12
?0001	0004	R00	9	15
?0002	000B	R00	13	

Macro Definitions

=====

[17]

The above listing is separated into four major sections. The following is a description of each section and item with a numeric reference to the above listing (e.g. [16]).

The Header section includes:

- [1] The Assembler version number
- [2] The time and date of the assembly
- [3] The source, listing and object file names
- [4] The command line option list

The Body section of the listing includes:

- [5] The source-file line number, in decimal
- [6] The load address for the code or data on that line, in hex
- [7] The actual code or data bytes, in hex
- [8] The source code, including comments

The Summary section of the listing includes:

- [9] The number of errors found in the module
- [10] The number of bytes of code plus constant data (FCBs) generated for the module, in decimal
- [11] The CRC (Cyclic Redundancy Code) checksum for the module. Note that the CRC does not only depend on the source code, but also on when the assembly was performed, and if the "S" option (or LOCSYM+ directive) was used.
- [12] The current module name, surrounded by "#" characters. Lastly, the Symbol and Cross-Reference Table section of the listing includes a list of user-defined symbols including:
- [13] Segments defined within the module (CODE, UDATA, etc.)
- [14] External symbols (those declared as EXTERN)

- [15] Public symbols (those declared as PUBLIC, such as main and c)
- [16] Local symbols (labels that are not declared PUBLIC or EXTERN, ?0000)
- [17] Macro names that are defined within the module

The following information is displayed for each symbol in the table:

Symbol

The symbol's user-defined name

Value

The value (address) of the symbol, relative to the beginning of the current segment, within the current module

Type

The symbol's type (see 5.9.16 LSTXRF for an explanation of this field)

Define

The source line where the symbol is defined

Refline

The source line(s) where the symbol is referenced

Please note that the "W" (wide) command line option and the "LSTWID +" assembler directive can be used to have the assembler put include-file nesting information in the Body section of the listing. See 5.9 Listing Directives for more information.

Macro generated lines will, if included in the listing, have their source line number fields replaced with a slash ('/'), as they do not represent actual lines of source code. This way, the numeric line numbers always show the actual number of source lines read from the input files. See section 7 for more information on macros.

2.9.4 Multi-Module Listings

The A6801 assembler has support for multiple modules within a single source file. This capability is described in the next section, Creating Assembler Programs.

The structure of a listing file produced from assembling a source file with multiple modules is slightly different from the previously-described listing:

- Each module within the source file produces its own Body, Summary and Symbol/Cross Reference sections in the listing. After the last source line in each module, the module's name is printed out surrounded by "#"'s in the Summary section -- making it easy to find a particular module in a long listing.
- At the end of the listing there is a Final Summary for the entire file that lists the total errors, bytes (in hex), and number of modules processed in the source file:

```
Errors:  None
Bytes:   52
Modules: 2
```

2.10 Downloading and Testing

This section discusses some of the issues involved in downloading and testing an 68HC11 program once it has been assembled and linked.

2.10.1 Overview

Once an A6801 assembler program has been assembled and linked, it must be tested and debugged. The methods employed to do this will vary greatly with your development situation. Typically your code will be tested and debugged using a simulator/debugger, an in-circuit emulator, or an EPROM programmer for the creation of EPROMs to be installed in your target hardware.

The following sections will briefly discuss each of these options.

2.10.2 Hardware Emulators

The Archimedes A6801 Kit supports most 68HC11 hardware emulators via a selection of XLINK output formats. Most emulators support symbolic debugging with A6801 programs (this allows you to refer to global symbols by name rather than by address).

The method employed to download a program to an emulator varies greatly with the manufacturer, so you must refer to their documentation for specific instructions (you can also refer to the EMULATOR.DOC file on the distribution diskettes).

In general, when creating A6801 programs that are to be debugged on a hardware emulator, be aware of the following:

- You should, if your emulator supports it (and most do), select one of the symbolic output formats available with XLINK, such as UBROF. You can always use the default Motorola format with most 68HC11 emulators, but this will not give you symbolic support.
- All symbols that you wish to make available to the emulator's symbol table should be made PUBLIC in your source file. Alternately, you can include ALL symbols in the object file by using the "S" assembler command line option or the "LOCSYM+" assembler directive within the source file.

2.10.3 PROM Programmers and Target Boards

Most EPROM programmers and 68HC11 development boards support direct downloading of absolute 68HC11 programs in the Motorola S-Record format. Only executable code and constant data can be placed in a Motorola file.

The procedures used to download Motorola files to EPROM programmers and target boards varies greatly with the manufacturer, so you must refer to their own documentation for specifics.

Quite often, an RS-232 link is used to communicate with your EPROM programmer or development board. If this is the case, you can use a general-purpose "terminal emulation" or communications program on your PC to talk to the device and to download Hex files to it. Examples of this type of program are Procomm (from -

Datastorm Technologies, also available as user-supported "shareware"), Crosstalk (DCA), Smartcom (Hayes), and many others.

3. Creating Assembler Programs

This section explains how to create assembly-language programs for the A6801 Assembler Kit, with examples presented along the way. It assumes that you have some experience with the 68HC11 (or similar) microcontroller and with assembly-language programming techniques in general.

While in the last section we covered assembler operation, here the emphasis is on overall program structure and various programming concepts (segments, multiple modules, libraries, etc.). Many of these subjects are covered in greater detail in the reference sections that follow (4-6), but the information presented here should get you started writing your own programs.

The following topics are covered in this section:

- Basic program structure, with an example source file
- Modules -- program and library
- Segments and segment directives
- Symbols and symbol directives
- Source line format
- Multi-module programming and libraries, with an example source file
- Miscellaneous topics, including using segments with the 68HC11 and include files

3.1 Assembler Source Files

First, we will look at the basic elements that go into an A6801 source file. Then we will look at an actual example program.

An A6801 assembly-language program can be built from one or more source files, which typically have a filetype of ".S07" (the default assumed by the Assembler), although any filetype can be used.

Each source file results in the generation of one or more object modules, which are functional groupings of program code or data, each with a unique name (see Appendix J: Glossary: Module). Modules are the basic building blocks that are used by XLINK when it is linking a program. The concept of modules and multi-module programming will be discussed in more detail later in this chapter. The source file or files that make up a single program are assembled and linked together to produce an absolute, 68HC11-executable program. (In section 2 we described the process of assembling and linking a complete sample program).

A source file can contain any number of source lines, each of which can be from 1 to 255 characters in length. Each source line falls into one of the following categories:

Blank line

A single <CR> (carriage return), or spaces or tabs ("white space") followed by <CR>.

Comment

A line beginning with a star (*) followed by any form of text, ending in a <CR>. Spaces or tabs may precede the comment:

```
* this is a comment line
```

Single Label

A single label used to symbolically reference that location. A colon (:) can be used to terminate a label, but is not required as long as there is a space or tab following the label. An optional comment can be placed on the same line if preceded by a semicolon. The label will be assigned the value (address) and type (absolute or relocatable) of the current Program Location Counter. See 4.3 User-defined Symbols for more information on symbols.

```
MY_LABEL:    Yes, this is a stupid example
```


Assembler Directives

Also known as "pseudo-ops", assembler directives are used to:

Designate module/program names and boundaries:

MODULE, NAME, ENDMOD, END

Declare segments and segment types:

ASEG, RSEG, COMMON, STACK, ORG

Assign values to symbols:

EQU, DEFINE, SET, =

Define/reserve space for data:

FCB, FDB, FQB, FCC, RMB

Designate symbol usage:

PUBLIC, EXTERN, LOCSYM

Control assembler listings:

LSTOUT, LSTXREF, PAGE, (12 more...)

Control conditional assembly:

IF, ELSE, ENDIF

Declare/specify macros:

MACRO, ENDMAC, %

Read "include" files into the source:

\$

See section 5: Assembler Directive Reference for complete information.

Source Statement

Consists of an optional label, followed by a 68HC11 source instruction and an optional comment.

LABEL: TBA

3.2 An Example Program

In this section we will look at an example 68HC11 Assembler source file and discuss comments, module and segment directives, symbols and instruction lines.

3.2.1 The ADEMO1 Source File

A small 68HC11 source file, ADEMO1.S07, is shown below.

```
1.* ADEMO1.S07 This is a short demo program for the A6801.
2.*           It takes the low-order nibble from the
3.*           BYTE_VAL and returns the ASCII equivalent in
4.*           A.
5.           NAME ASM_DEMO_1
6.           PUBLIC BYTE_VAL
7.           PUBLIC BINHEX
8.           RSEG CODE
9.           P68H11
10.BINHEX:
11.           LDAA BYTE_VAL    get parameter
12.           ANDA #$0F        mask off upper nibble
13.           CLC              clear carry bit
14.           SBCA #$10        subtract 10
15.           BCS LESS         to see if <10
16.           ADDA #'A'        >= 10, make it 'A'-'F'
17.           JMP DONE
18.LESS:
19.           ADDA #'0' + 10    <10 make it '0'-'9'
20.DONE:
21.           RTS
22.           RSEG UDATA      Now use segment UDATA
23.BYTE_VAL:
24.           RMB 1           Storage of one byte
25.           END            End the module and the file
```

This is a trivial routine that takes the byte at location `BYTE_VAL` (potentially put there by another routine that calls this one) and converts the low-order nibble (4-bits) to a single Hex-ASCII character ("0" - "F").

While a source file could theoretically consist of just a single source statement, the above program serves as a good typical example source file for our purposes here. However, not ALL of these elements MUST be included in EVERY file (those that are will be pointed out).

Now we will look at each group of lines in the program. Along the way we will discuss the concepts of comments, modules, segments, source lines, symbols and expressions.

3.2.2 Comments

Lines #1 - 4 consist of comment lines, which must start with a star (*) as the first non-tab or non-space character. Comments will appear in the output listing but are otherwise ignored by the Assembler.

3.2.3 Modules

NOTE

Although the A6801 assembler supports multi-module source files, our example program only contains one module for the sake of simplicity. See the ADEMO2.S07 file for an example of a program containing multiple modules. Line #5 in the example contains a module directive, NAME, which does two things for us:

1. It assigns a module name (ASM_DEMO_1) to the block of code and/or data that exists between it and the END statement (or between it and the next ENDMOD statement in a multi-module program). We can pick any legal name we want for the module name (see 3.2.5 Symbols for symbol-naming rules). The module name you assign does NOT have to be the same as the source file name, although it can be.

A NAME statement is optional -- if it is left out the assembler will default to using the source file name (ADEMO) as the module name.

2. Secondly, the NAME statement gives the named module a program attribute. This means that when the object file (ADEMO1.R07 in our case) is linked with XLINK, the module ASM_DEMO_1 will ALWAYS be loaded and will be part of the output file.

If instead of a NAME statement, we used the alternate MODULE statement, then the module ASM DEMO 1 would be given a library attribute. Modules that have a library attribute are only loaded by XLINK when they contain a PUBLIC (globally known) symbol that is referenced by another module.

Typically, this means that in order for ASM DEMO 1 to be loaded by XLINK as a library module, another module would have to have an EXTERN statement for a symbol that is PUBLICed in ASM_DEMO_1 (such as "BINHEX").

NOTE

The use of the library modules (and multi-module source files) is described in greater detail later in this section (3.3 Multi-Module Files and Libraries) as well as in the XLINK and XLIB Sections of the manual.

3.2.4 Segments

Line #8 of ADEMO.S07 contains a segment directive, RSEG, that tells the assembler to assign the code that follows (or data, if there were any) to a relocatable segment we have named "CODE". This segment would normally be linked at an address that corresponds to ROM in your target design.

Line #22 also contains an RSEG statement which assigns the data that follows to a segment we have named "UDATA". When we link ADEMO1, we will want to locate the segment UDATA at a valid RAM address for the 68HC11.

For additional comments on the 68HC11 Architecture, see 3.4.1 More About Segments and the 68HC11.

A segment, simply stated, is a user-created and named contiguous area of memory. With segment directives you can tell the assembler to place code (executable instructions) or constant data (FDB, FCB, FCC, or FQB directives) in a named segment, or you can have it reserve space for data in a named segment (using a RMB directive as in the example).

Note that a given segment name can be re-used over and over again within a module, or within a multi-module program (.e.g, multiple occurrences of "RSEG CODE" in a module is allowable). The code

from each occurrence of the segment definition will be placed successively in memory when it is loaded by the Linker.

It is important to mention here that the segments named "CODE" and "UDATA" in our example could have been named "BOB" and "RAY", or any legal name you like (see 3.2.5 Symbols for rules). How many segments there are, what they are named and how they are used in an all-assembly-language program is completely up to the programmer.

NOTE

The Archimedes C-6811 compiler DOES use certain pre-defined segment names in special ways (CODE, UDATA, ZVECT, etc.). If you are planning to write assembler code that interfaces with C refer to the C Compiler Section of this manual for more information.

In a small, 68HC11 assembly-language, embedded-controller application you will typically create two segments as we have done in our example: one for code and constant data linked to where your ROM is located; and one to reserve space for data, linked to where internal or external RAM is located.

In most applications you will be using only two of the four A6801 segment directives: RSEG and ASEG. The RSEG (relocatable segment) directive is used to put code or data into the specified segment whose address is determined at link-time by the XLINK Linker.

The ASEG (absolute segment) directive is used when you wish to fix the location of a section of code or data at assembly-time (the assembler, not the linker determines the final address). When ASEG is used, the actual location of an address is specified with the ORG assembler directive (see 5.4.6 ORG for more information). ASEG does not create a real "segment". In effect it is declaring "no segment".

Care should be taken when using ASEGs in your program as constant data or code created using ASEGs can overwrite code or data in an RSEG segment if the load addresses conflict. XLINK may not warn of such a condition.

NOTE

We strongly recommend that you use only relocatable segments in your programs. Symbol typing information required by some emulators is only available when relocatable segments are used.

It is a simple matter to place a grouping of code or data at any fixed address by creating a segment for it and locating that segment at the desired address at link time. This would be the case for locating interrupt vectors or for placing "setup" data at the address where a non-volatile RAM is located.

3.2.5 Symbols

Lines #6 and #7 both contain PUBLIC directives that make the symbols "BINHEX" (Line #10) and "BYTE_VAL" (Line #23) global -- that is, one that can be referenced by other modules. All that other module needs to do to call the routine "BINHEX" is to declare it as "EXTERN"; the same applies to accessing the data location "BYTE_VAL". The XLINK Linker then resolves all external addresses at link time:

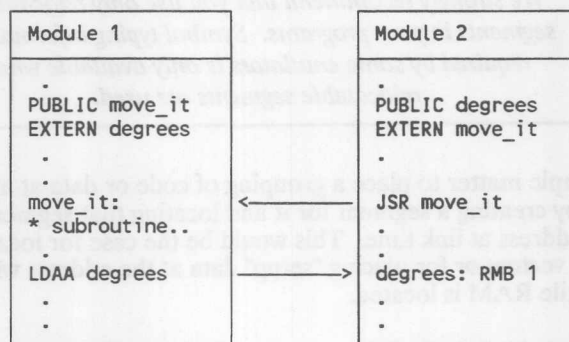
```
EXTERN  BINHEX
JSR     BINHEX    call BINHEX which resides in another module

EXTERN  BYTE_VAL
LDAA    BYTE_VAL  data defined in another module
```

Note that PUBLIC and EXTERN only support one symbol per line:

```
EXTERN  symbol1, symbol2,..    *** NOT ALLOWED !! ***
```

To further illustrate the use of global symbols, see the diagram below. Module 2 can call the routine "move_it" in Module 1, and Module 1 can access the variable "degrees" located in Module 2:



For more information on the use of PUBLIC and EXTERN, see 5.3 Symbol Directives.

Note the following rules regarding symbol use (also see section 4: Expression Reference):

- User-defined symbols: All user-defined symbols (labels, module names, segment names, etc.) can be up to 255 characters in length and ALL characters are significant both in the Assembler and the XLINK Linker. Upper/lower case IS significant for user-defined symbols:

My_own_symbol: and...

MY_own_symbol: are NOT EQUIVALENT !!!

- Pre-defined symbols: These include the 68HC11 instructions, the "*" (location counter) symbol, 68HC11 register names. These pre-defined symbols are NOT case-sensitive:

Also, note that you CANNOT re-define any of the pre-defined, internal 68HC11 symbols (e.g., PLC, program location counter)

- Characters: The first character in a label must be alphabetic (A-Z, a-z), or the characters "?" or " ". (The remainder of the label, if any, can contain any of the already mentioned characters or the decimal digits 0-9). The characters "*" and

"%" CANNOT be used in symbol names as "*" represents the location counter and "%" is used to designate macros.

3.2.6 Instruction Lines

Lines #11 - #21 consist of the 68HC11 source code (instruction) lines and form the body of the program. In general, source code lines must follow the general format of:

Label: Instruction Operand * Comment

or for assembler directive (pseudo-opcode) lines:

Label: Directive Operand Comment

The label, comment, and sometimes, the operand fields are optional.

<i>Note</i>
<i>A Label or Instruction may not be preceded by tabs or spaces (a label has to start in the first column)</i>
<i>If a Label is present, it must be separated from an Instruction or a Directive by a colon (:), spaces, or tabs.</i>
<i>Instructions and Operands may be separated by spaces or tabs</i>
<i>Operands must be separated by commas (,) only</i>
<i>Comments must be separated from instructions or operands by a star (*)</i>
<i>A source line must not exceed 255 characters</i>
<i>In listing files, tabs (ASCII 09H) are expanded to 8 positions (that is, to columns 9, 17, 25, etc...)</i>

Regarding constants, A6801 supports a wide range of types, such as:

'A'	ASCII constant
'ABC'	ASCII constant
'AB"C'	ASCII constant (AB'C)
78	Decimal constant
\$0F6	Hexadecimal constant
10110001B	Binary constant
457O	Octal constant
"1.0E3	IEEE 32-bit floating point constant

Note that hex constants that begin with the digits A-F must be preceded by a 0.

Line #19 contains a simple arithmetic expression that is evaluated by the assembler:

```
ADDA    #'0' + 10    < 10, make it '0'-'9'
```

A6801 supports a full set of operators that can be used in expressions:

Unary	-, .NOT., .HWRD., .LWRD., .LOW.
Arithmetic	+, -, *, /, .MOD.
Logical	.NOT., .AND. (or &), .OR., .XOR.
Comparison	.EQ. (=), .NE. (<>), .GE. (>=), .LE. (<=), .GT. (>), .UGT. (>>) and .ULT. (<<)
Bit-shift	.SHR., .SHL.
Special	.DATE., .SFB., .SFE.

The special operator .DATE. can be used to include date/time version information in the assembled code. The .SFB. and .SFE. directives return the address of the beginning and end of a segment (evaluated at link time).

NOTE

For complete details on using constants and expressions, please refer to section 4: Expression Reference.

3.2.7 The END Directive

Lastly, Line #25 in ADEMO.S07 contains an END directive, which marks the end of the module ASM_DEMO_1 as well as the end of the source file. The END statement is not optional, and should always be the last non-blank, non-comment line in the source file. The END directive can take a symbol as an optional parameter, which has the effect of tagging that symbol as the "program entry point":

```
END    program_start
```

where program_start is a label that is set to the address of the first instruction that is to be executed in the program. The program entry point is a special value that is passed on to the Linker and the output file.

The program entry point is used by some hardware emulators to determine where a program starts executing from. XLINK will yield a warning message if a program has more than one program entry point declared for it.

3.3 Multi-Module Files and Libraries

This section explains the creation and use of multi-module source files and libraries.

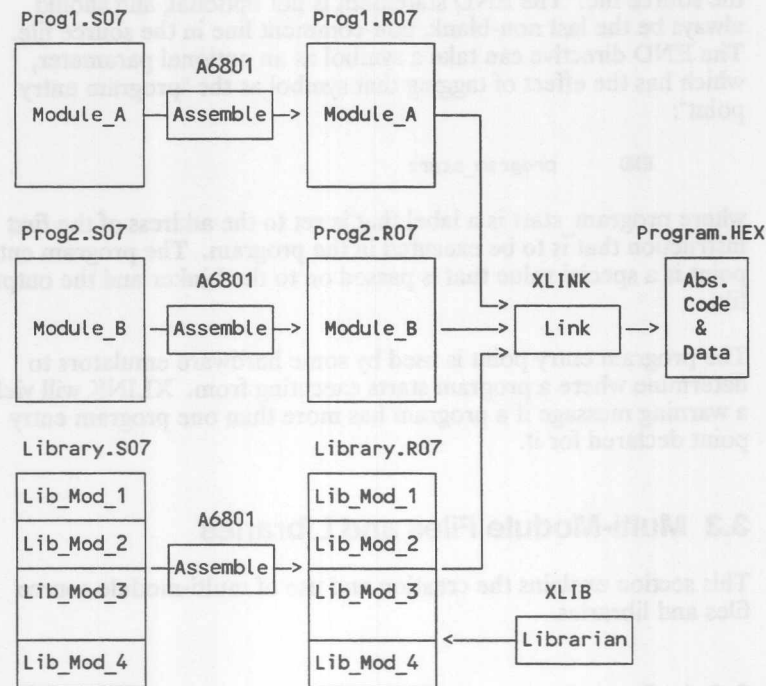
3.3.1 Overview

The A6801 Assembler supports the use of multiple modules within a single source file, as well as multiple source files in a single program. The main application for this feature is the creation of an object library file that contains a number of small modules, each of which can be loaded independently of the others by the XLINK linker (this avoids the problem of "monolithic" libraries where you get all or nothing when you load a library module).

This feature also helps reduce the number of source files that you need for an application since a single library file can contain as many modules as you want. You don't have to create separate source files for each module, which simplifies program maintenance.

3.3.2 A Library Application

The diagram below will help illustrate the structure of an Assembler program with a multi-module library file:



In the above diagram, the source files Prog1.S07 and Prog2.S07 represent the main source files for the application. Each of these files contain one module each (Module A, Module B). Each of these modules should be declared with the NAME assembler directive, indicating that they are "program" (vs. library modules).

The source file Library.S07 represents a library file to be used for the application. It consists of multiple modules, each of which is declared with the MODULE assembler directive, indicating that they are "library" modules.

These three .S07 source files are assembled separately to produce three .R07 object files, which are passed on to the linker. The XLINK linker will automatically load Module_A from Prog1.R07 and Module_B from Prog2.R07, as these are "program" attribute modules. However, XLINK will only load modules from the

Library.R07 file that have entries (PUBLICed labels) that are referenced (by an EXTERN directive) by a module that has already been loaded.

For example, Lib Mod 1 will only be loaded by the Linker if another module (e.g. Module A, which IS loaded) has an "EXTERN symbol" directive, where "symbol" is also PUBLICed in Lib_Mod_1.

Other than that they contain multiple modules, library source and object files are just like other assembler files (they use .S07 and .R07 filetypes, for example). The XLIB Librarian utility is used to manage library object files. It can insert and delete modules in a library, replace old modules with new ones, change a module's attribute from "program" to "library", etc.

3.3.3 Creating Multi-Module Files

The following general outline shows how a multi-module assembly-language source file is designed:

```
NAME/MODULE module_name
.
.      Module #1
.
ENDMOD
NAME/MODULE module_name
.
.      Module #2
.
ENDMOD
NAME/MODULE module_name
.
.      Last module
.
END
```

The NAME or MODULE directive is used to mark the beginning of a module. The ENDMOD directive then marks the end of the module, except when it is the last module in a file. The END directive must be used to terminate the last module in the source file.

As you recall, the NAME directive gives a module the "program" attribute while MODULE gives it a "library" attribute.

There must not be any source lines, except for comments and list control directives, prior to the first NAME/MODULE directive or between an ENDMOD and a NAME/MODULE directive. The

same applies between the END directive and the end of the source file.

The following rules apply to multi-module source files:

1. At the beginning of a new module all user symbols, except those that have been declared with the "DEFINE" directive, and MACROs, are deleted; the Program Location Counter is cleared; and the default relocation mode is set to "absolute" (ASEG) (A subsequent "RSEG seg_name" directive can be used to set the mode to relocatable).
2. The Assembler list control directives (see 5.9 Listing Directives) remain in effect throughout the assembly and are not affected by the module directives. Refer to 5.2 Module Directives, for more information on using the module directives.

3.3.4 Example Multi-Module File

The following program, ADEMO2.S07, is a simple example of a multi-module assembler program. It is based on the ADEMO1.S07 file explained earlier in the section:

```
* ADEMO2.S07 This is a multi-module demo file
*           for the A6801 assembler.

NAME ASM_DEMO_1 This module has a
*               "program" attribute.
PUBLIC BYTE_VAL These 2 symbols are
PUBLIC BINHEX    located in this module,
EXTERN PUTCHAR   While this one is in another.
RSEG CODE        Executable code segment.
P68H11           Enable 68HC11 specific instructions

BINHEX:
LDAA BYTE_VAL
ANDA #$0F
CLC
SBCA #$10
BCS LESS
ADDA #'A'
JMP DONE
LESS:
ADDA #'0' + 10
DONE:
JSR PUTCHAR
RTS
```

```

RSEG UDATA                      Uninitialized Vars. segment
BYTE VAL:
RMB 1
ENDMOD                          End 'program' module ASM_DEMO_1

MODULE PUTCHAR                  This module has a 'library'
PUBLIC PUTCHAR                  attribute.
P6801                           Enable 6801 specific instructions
RSEG CODE
PUTCHAR:
TBA
JSR $E3B3
RTS
END                              End library module PUTCHAR, and end the
                                entire file.

```

This program is an expansion of the ADEMO1.S07 file we looked at earlier in this section. It simply prints out a string of predefined bytes in hexadecimal form on the 68HC11 serial port. Refer to the comments in the source code for details.

There are two modules in this file: "ASM_DEMO_1", a "Program" module and PUTCHAR, a "Library" module. The code in the first module calls the PUTCHAR entry in the second, which prints out the ASCII character converted by the BINHEX routine. Because the first module includes an EXTERN statement for this entry, the second (Library) module will be loaded by XLINK at link time (PUTCHAR is declared as PUBLIC in the second module).

A real user-library file would contain dozens of modules instead of just the two we have shown here. Library files can also be created at the object-file level by combining multiple .R07 modules into a single file by using the XLIB Librarian -- see The XLIB Librarian chapter of this manual.

3.4 Additional Assembler Topics

This section covers miscellaneous topics including notes on using segments with the 68HC11, and "include" files.

3.4.1 More About Segments and the 68HC11

<i>NOTE</i>
<i>For an overview of the 68HC11 microcontroller architecture, refer to a Motorola M68HC11 Reference Manual.</i>

When creating assembler programs for the 68HC11 (and proliferation chips), you usually want to place executable code and constant data in the 68HC11's "code space", in ROM or EPROM. Space for data elements (variables, buffers, etc.) is usually reserved in the 68HC11's "internal data space" (on-chip RAM) or in its "external data space", (external, off-chip RAM).

However, segments are not strictly differentiated as being "code", "internal data" or "external data" segments, they are just plain old segments that get relocated by the Linker to an address you define (see the "-Z" XLINK command in section 4.3 XLINK Command Detail in the Linker chapter).

<i>NOTE</i>
<i>The segment "Type" parameter, which is set to CODE or DATA in the XLINK -Z command, does not affect how XLINK locates segments, it is used only to provide symbol typing information for the symbolic output formats.</i>

What makes a segment a "code" segment is that it contains executable code and it is linked at a 68HC11 address where there is ROM that is decoded as program memory (although in some cases the hardware may be designed so that RAM gets decoded as program memory).

What makes a segment a "data" segment is that it is used to reserve space for variables, buffers, etc. (not constant data) and that it is linked at an address that corresponds to 68HC11 RAM space.

Segments that contain executable code or constant data must be linked to where your program memory (usually ROM) is located. Segments that contain reserved data locations must be linked to addresses that correspond to where on-chip or off-chip RAM is located.

Normally, constant data directives should NOT be placed in a non-constant data segment that is to be linked to an internal or external RAM address. This is because these directives cause constant initializers to be sent to the output file, and data memory cannot be initialized in this manner.

3.4.2 Include Files

The "\$" symbol is an assembler directive which enables the user to include files in the source code. The dollar sign must be located in the first column of a source line, and no extra characters are allowed between the \$ and the filename:

```
$PDEFS.INC          include program defines
```

The include file names follow the same rules as source files specified in the assembler start command and should be terminated with a <CR>. If a filetype is not explicitly specified, ".S07" is assumed as the default. A filetype of ".INC" is often used for include files.

Please refer to 5.8 Source Code Expansion Directives for more information on "include" files.

4. Assembler Expression Reference

This section is a reference to the use of expressions in the A6801 Assembler. Expressions are made up of operands (constants, user-defined symbols and pre-defined symbols) and operators (add, subtract, compare, etc.), so we will also cover these important Assembler elements as well (see Appendix J:Glossary for a definition of terms).

The list of topics covered in this section includes:

- Constants (integers, ASCII characters and real numbers)
- User-defined symbols (labels, EQUated values, etc.)
- An overview of expression formation, including relocatable expressions and external expressions
- Summary listing of operators grouped by function type and order of precedence (unary, arithmetic, logical, etc.)
- Detailed reference of all A6801 operators, listed alphabetically

4.1 Constants

This section describes the three types of constants that can be used in an A6801 Assembler program: Integers, ASCII characters and Real numbers.

4.1.1 Integer Constants

Note the following guidelines regarding the use of integer constants in your programs:

- Integer constants may range in value from -2,147,483,648 to +2,147,483,647 (they are stored internally in the Assembler as signed 32-bit numbers)
- Constants are written as a sequence of digits with an optional "-" (minus) sign in front to indicate a negative number
- Commas and decimal points are not permitted
- Constants may be specified in any of four number bases:

Base	Prefix/Suffix	Example
Decimal (base 10)	(none)	1234
Hexadecimal (base 16)	\$(P)	\$0F301
Binary (base 2)	B(S)	11011011B
Octal (base 8)	O(S)	6310

Note that if a hexadecimal constant begins with the digits 'A' through 'F', a leading zero must be used before the first digit so the Assembler will not mistake the constant for a symbol.

The following are examples of integer constants:

123	decimal
\$23	hexadecimal (= 291 decimal)
\$0F2	hexadecimal (= 242 decimal)
1230	octal (= 83 decimal)
110B	binary (= 6 decimal)
-123	decimal, negative number

4.1.2 ASCII Character Constants

Note the following guidelines regarding the use of character constants in your programs:

- A6801 allows 1, 2, 3 or 4 ASCII characters to be stored as a single constant (this is because character constants are stored internally in the Assembler in a 32-bit format).
- Each constant (of 1 - 4 characters) must be surrounded by single quotation marks (''). To enter a single quote character (') as part of a constant, use two single quotes in a row (''). The following are examples of ASCII character constants:

'A'	1-character constant: A
'AB'	2-character constant: AB
'ABCD'	4-character constant: ABCD
'A''CD'	constant with a single quote in the middle: A'CD
'A'''	constant with a single quote at the end: A'
'''A'	constant with a single quote in the beginning: 'A
''	an empty character constant (no characters)

- Only printable characters and spaces (those with codes between 32 and 127 decimal) may be used in ASCII constants in the Assembler

4.1.3 Real Number Constants

The A6801 Assembler will accept real numbers as constants and convert them into IEEE single precision (signed 32-bit) real number format for data initialization purposes (see 5.6.5 FQB for examples of this feature).

NOTE

You cannot use real numbers in expressions as the assembler stores them internally as signed 32-bit integers (see 4.4 Expressions). Real numbers may only be used for data initialization purposes as described for the FQB directive in section 5.6.5.

Note the following guidelines for using real number constants in assembler programs:

- To specify a real number, precede it with a double quote character (") and format the number using the following exponential notation:
`"<sign> <mantissa> e <exponent sign> <exponent>`
- An optional decimal point can be used in the mantissa
- Note that spaces and tabs are not allowed within a real number constant

Examples of real number constants:

"123	123.0, integer mantissa
"123e1	1,230.0, positive exponent
"-123e1	-1,230.0, negative number
"0.001	0.001, decimal mantissa
"100e-5	0.001, negative exponent

4.3 User-defined Symbols

The following guidelines apply to the creation of user-defined symbols in A6801 Assembler source files:

- User-defined symbols may be up to 255 characters long and all characters are significant in distinguishing one symbol from another (this is true for the XLINK Linker also).
- Symbols must begin with an alphabetic character, an underscore character (_) or a question mark (?); they may NOT begin with a number. The remainder of the symbol

name can include the digits 0 through 9 in addition to the aforementioned characters. The characters "*" and "%" CANNOT be used in symbol names as "*" represents the location counter and "%" is used to designate macros.

- Upper and lower case characters are considered different in user-defined symbols (as opposed to A6801 pre-defined symbols where no distinction is made).
- User defined symbols may not be given the same name as any of the pre-defined symbols (i.e., those used for instructions, registers, etc.). In other words, you cannot override the value assigned by the Assembler to a pre-defined symbol.
- The maximum number of symbols which can be used in a source file depends upon available memory and the length of the symbol names.
- The maximum number of externally-declared symbols (i.e., symbols declared as EXTERN) in a module is limited to 256. The number of public symbols (those declared as PUBLIC) in a module is limited only by memory and the length of the symbol name. See 5.3.2 PUBLIC and 5.3.3 EXTERN for more information.

4.4 Expressions

This section discusses how expressions are formed and used in an Assembler program and contains a reference to all of the A6801 operators.

4.4.1 Expression Basics

An expression is a combination of operands (constants, pre-defined symbols or user-defined symbols) and operators (+, -, .GE., .XOR., .NOT., etc.) which together yield a value at the time a program is assembled (this is called assembly-time evaluation). Operands are the items upon which operations are to be performed while Operators specify the actual operation.

Operands

The following items may be used as operands in Assembler expressions:

Constants: See 4.1 Constants for a description of integer, character and real-number constants. A constant used as an operand in an expression takes on the 32-bit integer value associated with the constant.

NOTE

You cannot use real (floating point) number constants in expressions as the Assembler stores them internally as signed 32-bit integers. Real numbers may only be used for data initialization purposes (see 5.6.5 FQB).

Labels: Labels are a form of user-defined symbol (see 4.3 User-Defined Symbols). When a label is used as an operand in an expression it takes on the value of the Program Location Counter when the label is assembled.

Defined Values: User-defined symbols that are created with the EQU, SET and DEFINE Assembler directives may be used as operands. These directives are described in 5.5 Value Directives.

*** (star sign):** When this special symbol is used as an operand it takes on the value of the Program Location Counter when the expression is evaluated.

Operators

There are two basic types of operators, Unary and Binary. Unary operators take one operand following the operator. Binary operators take two operands, one before and one after the operator. The following is an example of an expression involving a unary operator:

```
Value      EQU      .NOT. Symbol1
```

where .NOT. (bitwise negation) is a unary operator and Symbol1 (a user-defined symbol) is its operand.

The following is an example of a simple expression using a binary (two-operand) operator:

```
Data1      EQU      Value .OR. 1000000B
```

where the operands are "Value" (a user-defined symbol) and "10000000B" (a binary constant), and the operator is ".OR." (bitwise logical OR).

See sections 4.4.7 Summary Table of Operators and 4.5 Operator Reference for a complete list of operators supported by A6801.

4.4.2 Delimiters in Expressions

For expressions involving "symbolic" operators (+, /, >=, <=, etc.), spaces and tabs are optional and have no effect on the evaluation of expressions. For example, the following two expressions are equivalent:

```
((1 + 2) * LENGTH / HEIGHT) .MOD. 5  
((1+2)*LENGTH/HEIGHT) .MOD. 5
```

You may use spaces and tabs to improve the readability of your expressions or leave them out to make expressions very compact.

Note, however, that spaces or tabs are required as delimiters when using the "alphabetic" operators (.NOT., .SHL., .GE., etc.). For example, the space between the operator ".MOD." and the constant "5" in the above expressions is required because without it, the four characters ".MOD.5" would be interpreted as a legal user-defined symbol and not an operator-operand combination. (See the ".MOD." operator in 4.5 Operator Reference).

Spaces and tabs should not be placed within symbols.

4.4.3 TRUE and FALSE in Expressions

When an expression interprets an operand as either true or false, the following rules are used:

- If an operand's value equals zero, it is considered as FALSE
- If an operand has a non-zero value, it is considered as TRUE

When an expression evaluates to FALSE the Assembler generates a value with all "zeros". A TRUE expression yields a value with all "ones" so that it is equal to...

\$0FF	for byte values
\$0FFFF	for word values
\$0FFFFFFFF	for double-word values

4.4.4 Relocatable Expressions

Expressions that involve 68HC11 addresses result in values that are either absolute (completely resolved at assembly-time) or relocatable (not fully resolved at assembly time because the Linker determines where segments are actually located in memory).

There are some restrictions to the use of relocatable expressions in Assembler programs. This is because the Linker can only add offsets when it is attempting to resolve relocatable expressions:

- Relocatable operands (e.g., user-defined symbols) can only be added in an expression
- When two relocatable operands are involved in an expression, they must both belong to the same relocatable segment (e.g., they both must belong to the segment named CODE) and they may only be subtracted (yielding an absolute value).

- The following list of expression combinations involving relocatable operands show which ones are legal and which ones are not. The type (relocatable, absolute or invalid) of the resulting value is also shown:

Operand types / Operator	Resulting value
Relocatable + Absolute	Relocatable
Relocatable - Absolute	Relocatable
Absolute + Relocatable	Relocatable
Absolute - Relocatable	INVALID expression
Relocatable + Relocatable	INVALID expression
Relocatable - Relocatable	Absolute

4.4.5 External expressions

There is a limit to the complexity of expressions involving external symbols (symbols that are defined outside of the current module). This is because the Linker can only add offsets when it is attempting to resolve relocatable expressions:

- Symbols that are declared as EXTERN in a module can only be added (the "+" operator used) in an expression; other operations involving external symbols are NOT allowed

4.4.6 Operator Precedence

Each A6801 operator has a precedence number assigned to it which determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, or first evaluated) to 7 (the lowest precedence, or last evaluated). See the next section for a summary table of all the operators grouped by precedence and function.

The following rules determine how expressions are evaluated:

- The highest precedence (lowest number) operators are evaluated first, then the next highest precedence operators, and so on until the lowest precedence operators are evaluated.
- Operators of equal precedence are evaluated from left to right in the expression.

- Parentheses () can be used to group operators and operands and to control the order in which expressions are evaluated. For example, the expression...

8 / ((1 + 1) * (2 + 2)) evaluates to 2

4.4.7 Summary Table of Operators

Unary Operators -- Precedence 1 (Highest Precedence)

-	Unary minus
.NOT.	Logical NOT
.LOW.	Low byte of word
.HIGH.	High byte of word
.LWRD.	Low word of double word
.HWRD.	High word of double word
.DATE.	Current time and date
.SFB.	Segment beginning address
.SFE.	Segment ending address

Exponential Arithmetic Operators -- Precedence 2

(A6801 does not support any exponential operators)

Multiplicative Arithmetic Operators -- Precedence 3

*	Integer multiplication
/	Integer division
.MOD.	Integer modulus
.SHR.	Shift right
.SHL.	Shift left

Additive Arithmetic Operators -- Precedence 4

+	Integer addition
-	Integer subtraction

Multiplicative Logical Operator -- Precedence 5

.AND.	Logical AND
-------	-------------

Additive Logical Operators -- Precedence 6

.OR.	Logical OR
.XOR.	Logical exclusive OR

Comparison Operators -- Precedence 7 (Lowest Precedence)

.EQ. or =	Equality
.NE. or <>	Inequality
.GT. or >	Greater than (signed integers)
.LT. or <	Less than (signed integers)
.UGT. or >>	Greater than (unsigned integers)
.ULT. or <<	Less than (unsigned integers)
.GE. or >=	Greater than or equal (signed integers)
.LE. or <=	Less than or equal (signed integers)

4.5 Operator Reference

This section consists of a detailed reference to all of the assembler operators in A6801. The operators are listed alphabetically (the previous section contains a summary of operators grouped by function and precedence).

- Unary Arithmetic Negation

The "-" unary minus operator performs arithmetic negation on a single operand to its right. There must not be any space or tab between the minus sign and the operand.

The operand is interpreted as a 32 bit signed integer and the result of the operator is the 2's complement negation of that integer.

+ Addition

The "+" addition operator produces the sum of the two operands which surround it. The operands are taken as signed 32 bit integers and the result is also a signed 32 bit integer.

Examples:

92 + 19 evaluates to 111
-2 + 2 evaluates to 0
-2 + -2 evaluates to -4

-**Subtraction**

The "-" subtraction operator produces the difference when the second operand (the one on the right) is taken away from the first operand (the one on the left). The operands are taken as signed 32 bit integers and the result is also a signed 32 bit integer.

Examples:

92 - 19 evaluates to 73
-2 - 2 evaluates to -4
-2 - -2 evaluates to 0

*******Multiplication**

The "*" multiplication operator produces the product of the two operands which surround it. The operands are taken as signed 32 bit integers and the result is also a signed 32 bit integer.

Examples:

2 * 2 evaluates to 4
-2 * 2 evaluates to -4
RECORD_SIZE * NUM_ENTRIES evaluates to bytes needed

/**Division**

The "/" division operator produces the quotient of the first operand (the one on the left) divided by the second operator (the one on the right). Any remainder is discarded (see the MOD operator). The operands are taken as signed 32 bit integers and the result is also a signed 32 bit integer.

Examples:

8 / 2 evaluates to 4
 -12 / 3 evaluates to -4
 256 - 28799 / BAUD_RATE evaluates to TH1 reload

.AND. (or &)**Bitwise Logical AND**

.AND. is an operator that performs bitwise AND between the two integer operands which surround it. For each bit position, the result is according to:

1 .AND. 1 = 1
 1 .AND. 0 = 0
 0 .AND. 1 = 0
 0 .AND. 0 = 0

Example:

11001100B .AND. 10101010B = 10001000B

.DATE.**Today's Date**

The .DATE. operator accepts a single operand to its right. The operand must evaluate to an integer from 1 to 6. It evaluates to an element of the current date and time of assembly according to the operand given as follows:

.DATE. 1 evaluates to the current seconds (0..59)
 .DATE. 2 evaluates to the current minutes (0..59)
 .DATE. 3 evaluates to the current hour (0..23)
 .DATE. 4 evaluates to the current day (1..31)
 .DATE. 5 evaluates to the current month (1..12)
 .DATE. 6 evaluates to the current year MOD 100

Example:

	RSEG	CONSTANTS
CODATE	FCB	.DATE. 5, .DATE. 4, .DATE. 3

The above code will store the assembly date in the segment CONSTANTS.

.EQ. (or =)**Equal To**

The .EQ. equality operator evaluates to TRUE if the two operands which surround it are identical in value or to FALSE (zero) if the two operands which surround it are not identical in value.

Examples:

```
1 .EQ. 2 evaluates to FALSE
2 = 2 evaluates to TRUE
'ABC' .EQ. 'ABCD' evaluates to FALSE
```

.GE. (or >=)**Greater Than or Equal To**

The .GE. "signed greater than or equal" operator evaluates to TRUE if the first operand (the one to the left) is equal to or has a higher numeric value than the second operand (the one to the right).

Examples:

```
1 .GE. 2 evaluates to FALSE
2 >= 1 evaluates to TRUE
1 .GE. 1 evaluates to TRUE
```

.GT. (or >)**Greater Than**

The .GT. "signed greater than" operator evaluates to TRUE if the first operand (the one to the left) has a higher numeric value than the second operand (the one to the right).

Examples:

```
-1 .GT. 2 evaluates to FALSE
2 > 1 evaluates to TRUE
1 .GT. 1 evaluates to FALSE
```

.HIGH.**Hi-Order Byte of Integer**

The .HIGH. operator takes a single operand to its right. The operand is interpreted as an unsigned, 16 bit integer value. The result is the unsigned 8 bit integer value of the higher order byte of the operand.

Example:

.HIGH. \$0ABCD evaluates to \$0AB

.HWRD.**Hi-Order Word of Double Word**

The HWRD operator takes a single operand to its right. The operand is interpreted as an unsigned, 32 bit integer value. The result is the unsigned 16 bit integer value of the higher order word of the operand.

Example:

.HWRD. \$0ABCDEF89 evaluates to \$0ABCD

.LE. (or <=)**Less Than or Equal To**

The .LE. "signed less than or equal" operator evaluates to TRUE if the first operand (the one to the left) has a lower numeric value than the second operand (the one to the right).

Examples:

1 .LE. 2 evaluates to TRUE
2 <= 1 evaluates to FALSE
1 .LE. 1 evaluates to TRUE

.LOW.**Low-Order Byte of Integer**

The .LOW. operator takes a single operand to its right. The operand is interpreted as an unsigned, 16 bit integer value. The result is the unsigned 8 bit integer value of the lower order byte of the operand.

Example:

.LOW. \$0ABCD evaluates to \$0CD

.LT. (or <)**Less Than**

The .LT. "signed less than" operator evaluates to TRUE if the first operand (the one to the left) has a lower numeric value than the second operand (the one to the right).

Examples:

```
-1 .LT. 2 evaluates to TRUE
2 < 1 evaluates to FALSE
2 .LT. 2 evaluates to FALSE
```

.LWRD.**Low-Order Word of Double Word**

The .LWRD. operator takes a single operand to its right. The operand is interpreted as an unsigned, 32 bit integer value. The result is the unsigned 16 bit integer value of the lower order word of the operand.

Example:

```
.LWRD. $0ABCDEF89 evaluates to $0EF89
```

.MOD.**Modulus Division**

The .MOD. (modulo) operator produces the remainder from the integer division of the first operand (the one on the left) by the second operand (the one on the right). The operands are taken as signed 32 bit integers and the result is also a signed 32 bit integer.

Examples:

```
2 .MOD. 2 evaluates to 0
12 .MOD. 7 evaluates to 5
3 .MOD. 2 evaluates to 1
```

.NE. (or < >)**Not Equal To**

The .NE. inequality operator evaluates to FALSE (zero) if the two operands which surround it are identical in value or to TRUE if the two operands which surround it are not identical in value.

Examples:

```
1 .NE. 2 evaluates to TRUE
2 <> 2 evaluates to FALSE
'A' .NE. 'B' evaluates to TRUE
```

.NOT.**Bitwise Logical Negation**

.NOT. is a unary operator which performs bitwise logical negation in the single operand to its right. For each bit position, the result is according to:

```
.NOT. 1 = 0
.NOT. 0 = 1
```

Examples:

```
.NOT. $01           = $FE
.NOT. $0001         = $FFE
.NOT. $00000001     = $FFFFFFE
.NOT. 01001001B     = 10110110B
```

.OR.**Bitwise Logical OR**

The .OR. operator performs bitwise OR between the two integer operands which surround it. For each bit position, the result is according to:

```
1 .OR. 1 = 1
1 .OR. 0 = 1
0 .OR. 1 = 1
0 .OR. 0 = 0
```

Examples:

```
11001100B .OR. 10101010B = 11101110B
```

.SHR.**Bit Shift Right****NOTE:**

The bit shift operators .SHR. and .SHL. interpret the first operand (the one on the left) as a 32 bit integer and the second operand (the one on the right) as an integer value between 0 and 32 (because there are only 32 bit positions which can be shifted). The first operand is bit shifted by the number of bits given by the second operand. Bits which are shifted in are zero bits.

The .SHR. operator shifts the first operand to the right as indicated by the second operand.

Examples:

01110000B .SHR. 3 evaluates to 0001110B
1111111111111111B .SHR. 20 evaluates to 0
14 .SHR. 1 evaluates to 7

.SHL.**Bit Shift Left**

The .SHL. operator shifts the first operand to the left as indicated by the second operand.

Examples:

00011100B .SHL. 3 evaluates to 11100000B
0000011111111111B .SHL. 5 evaluates to 1111111111100000B
14 .SHL. 1 evaluates to 28

.SFB.**Segment Beginning Address**

The .SFB. operator accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the absolute address of the first byte of that segment. This evaluation takes place at linking time.

The Archimedes C compiler uses this operator (and .SFE. below) to determine the extents of the ROM and RAM data segments so that it can properly initialize the RAM data area.

Example:

```

1 0000          NAME  .SFB.DEMO
2 0000          RSEG  CODE
3 0000 900000   ENTER: LDD  #.SFB. CODE
4              *      ...      ...

```

Even if the above code is linked with many other modules, accumulators A/B will still be properly loaded with the address of the first byte of the segment.

.SFE.

Segment Ending Address

The .SFE. operator accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the absolute address of the last byte of that segment. This evaluation takes place at linking time.

The Archimedes C compiler uses this operator (and .SFB. above) to determine the extents of the ROM and RAM data segments so that it can properly initialize the RAM data area.

Example:

```

1 0000          NAME  .SFE.DEMO
2 0000          RSEG  CODE
3 0000 900000   ENTER: LDD  #.SFE. CODE
4              *      ...      ...

```

Even if the above code is linked with many other modules, accumulators A/B will still be properly loaded with the address of the last byte of the segment.

.UGT. (or > >)

Unsigned Greater Than

The .UGT. "unsigned greater than" operator evaluates to TRUE if the first operand (the one to the left) has a larger absolute value than the second operand (the one to the right).

Examples:

```

2 .UGT. 1 evaluates to TRUE
-2 .UGT. 1 evaluates to TRUE
-1 .UGT. 1 evaluates to TRUE
(-1 .GT. 1 evaluates to FALSE)

```

.ULT. (or < >)**Unsigned Less Than**

The .ULT. "unsigned less than" operator evaluates to TRUE if the first operand (the one to the left) has a smaller absolute value than the second operand (the one to the right).

Examples:

```

1 .ULT. 2 evaluates to TRUE
-1 .ULT. 2 evaluates to TRUE
(-1 .LT. 2 evaluates to TRUE)

```

.XOR.**Bitwise Exclusive OR**

The .XOR. operator performs bitwise .XOR. on the two operands which surround it. For each bit position, the result is according to:

```

1 .XOR. 1 = 0
1 .XOR. 0 = 1
0 .XOR. 1 = 1
0 .XOR. 0 = 0

```

Examples:

```

11001100B .XOR. 10101010B = 01100110B
00001111B .XOR. 10101010B = 10100101B

```


5. Assembler Directive Reference

This section describes the directives which allow you to specify varying modes of operation of the Archimedes A6801 Cross Assembler. These descriptions include:

- A summary of all the Assembler Directives
- Module Directives
- Symbol Directives
- Segment Directives
- Value Directives
- Data Directives
- Conditional Assembly Directives
- Source Code Expansion Directives
- Listing Directives

5.1 Summary of Assembler Directives

This section lists all of the Assembler Directives with their syntax and a brief descriptive comment.

Module directives

NAME <symbol>	begins a program module
MODULE <symbol>	begins a library module
ENDMOD <expression>	ends a module
END <expression>	ends a source code file

Symbol Directives

PUBLIC <symbol>	other modules now know <symbol>
EXTERN <symbol>	<symbol> defined in other module
LOCSYM+,LOCSYM-	make all symbols public

Segment directives

ASEG	begin an absolute segment
ORG <expression>	specify the location counter
RSEG <name>[(align)]	begin a relocatable segment
STACK <name>[(align)]	begin a stack segment
COMMON <name>[(align)]	begin a common segment

Value directives

<label> EQU <expression>	a module-wide value
<label> SET <expression>	a redefinable value
<label> DEFINE <expression>	a file-wide value
<label> = <expression>	alternate spelling for EQU

Data directives

RMB <expression>	Reserve uninitialized memory byte
FCB <byte value(s)>	Define initialized bytes
FDB <word value(s)>	Define initialized words
QFB <dword value(s)>	Define initialized dwords
FCC '<string>'	Define constant string

Conditional Assembly directives

IF <condition>	begin conditional assembly block
ELSE	alternate conditional polarity
ENDIF	end conditional assembly block

Source Code Expansion Directives

\$<include file name>	insert an include file
MACRO %<name>	begin a macro definition
ENDMAC	end a macro definition
%<name>[<parameters>]	invoke a macro expansion

Listing Directives

LSTOUT+, LSTOUT-	turn listing on off
LSTCND+, LSTCND-	conditional assembly list on off
LSTCOD+, LSTCOD-	abbreviate initialized code list
LSTMAC+, LSTMAC-	macro definition list on off
LSTEXP+, LSTEXP-	macro expansion list on off
LSTWID+, LSTWID-	include nesting list on off
LSTFOR+, LSTFOR-	reformat listing lines on off
LSTPAG+, LSTPAG-	paginate listing on off
PAGE	immediate page break command
TITL '<string>'	title pages with <string>
PTITL '<string>'	PAGE plus TITL
STITL '<string>'	subtitle pages with <string>
PSTITL '<string>'	PAGE plus STITL
LSTXRF	cross reference table in listing
PAGSIZ <expression>	page break after <expression> lines

5.2 Module Directives

This section details the Assembler Directives that are used to name and declare modules within source files.

5.2.1 Overview of Module Directives

Module directives are used to mark the beginning and ending of source program modules and to assign names and types to them.

NOTE

Please see section 3: Creating Assembler Programs for an overview of using modules. You should also review the Linker and the Librarian sections of this manual for additional information regarding module use and management.

A single program source file (.S07 filetype) can be divided into more than one module. Each module can be "Program" type or "Library" type. Typically, a module is specified as a "Library" type if it is to be placed into a library of modules and accessed by the Linker from the library. It is specified as "Program" type if it is to be input directly to the linker as a separate relocatable module. The two different types may be mixed within a single source code file.

In the simplest case, a source code file can also consist of just a single module.

The following items are considered "global" to all the modules within a single source file (i.e., these items do not need to be re-defined within each module):

- Macro definitions
- Values set by the DEFINE directive
- All listing directives

All other items are considered to be "local" to the module where they are declared (i.e., these items must be re-defined with each module):

- Values set with the EQU or SET directives
- Labels
- Segments
- The Program Location Counter

5.2.2 NAME

Syntax: [<label>] NAME <module name>

The NAME directive marks the beginning of a "Program-type" module. It also assigns a <module name> to the module for future reference by the Linker and Librarian.

The NAME directive must be the first non-comment line of source code in a module. If a NAME directive is not specified, the Assembler will assign the name of the source file as the module name.

5.2.3 MODULE

Syntax: [<label>] MODULE <module name>

The MODULE directive marks the beginning of a "Library-type" module. It also assigns a <module name> to the module for future reference by the Linker and Librarian.

The MODULE directive must be the first non-comment line of source code in the module.

5.2.4 ENDMOD

Syntax: [`<label>`] ENDMOD [`<expression>`]

The ENDMOD directive marks the end of a module within a source code file that contains multiple modules (the END directive is used instead if there is only a single module in the source file). ENDMOD is used whether the module starts with a NAME directive or a MODULE directive.

The optional `<expression>` can be used to specify a label or address value as the program entry point. The program entry point is a special global symbol used by some hardware emulators to determine where a program starts executing from. If you specify a label which corresponds to the beginning (entry point) of your program, the Linker will output that symbol as the program entry point for use by an emulator. Otherwise, this optional expression serves no purpose within the program.

There can be only one program entry point defined for a program. The Linker will issue a warning if a second one is specified. The program entry point may also be specified in the END directive, described in the next section.

5.2.5 END

Syntax: [`<label>`] END [`<expression>`]

The END directive marks the end of the source file and the end of the last module defined in the source file (whether that module starts with a NAME or a MODULE directive).

The optional `<expression>` can be used to specify a label or address value as the program entry point. The program entry point is a special global symbol used by some hardware emulators to determine where a program starts executing from. If you specify a label which corresponds to the beginning (entry point) of your program, the Linker will output that symbol as the program entry point for use by an emulator. Otherwise, this optional expression serves no purpose within the program.

There can be only one program entry point defined for a program. The Linker will issue a warning if a second one is specified. The program entry point may also be specified in the ENDMOD directive, described in the previous section.

Module Directive Example

As an example of module directive use, refer to the following source lines:

NAME	DO_THIS	beginning of 1st module (Program type)
ENTER1:	LDAA #10	beginning of 1st subroutine
.		
.		
RTS		
TASK1:	LDAB #1	beginning of 2nd subroutine
.		
.		
RTS		
ENDMOD ENTER1		end of the 1st module also specifies program entry point
MODULE	DO_THAT	beginning of 2nd module (Library type)
ENTER2:	LDAB #10	
.		
.		
RTS		
END		end of 2nd module and source file

When the above source file is assembled, the Assembler will produce a single relocatable object file (.R07 filetype) which contains two relocatable object modules. The module named "DO_THIS" will have the "Program" module type and the module named "DO_THAT" will have the "Library" module type. The symbol "ENTER1" will be passed to the Linker as the program entry point as it is specified as a parameter for the ENDMOD statement.

5.3 Symbol Directives

This section covers the Symbol Directives PUBLIC, EXTERN and LOCSYM, which control how symbols are shared between modules.

5.3.1 Overview of Symbol Directives

There are two kinds of symbols that can be used within a module: local and global. Local symbols are only defined for the module in which they are declared. Global symbols are available throughout all the modules and source files that make up a program.

Local symbols in different modules may use the same symbol name. However, global symbols must be uniquely named throughout an entire program.

By default, all Assembler symbols are local. A symbol is made global by the use of the PUBLIC directive. Other modules may refer to this global symbol by declaring it as EXTERN. Multiple modules may reference a global symbol with the EXTERN directive but only one module can define it with the PUBLIC directive.

Please see 3.2.5 Symbols for a discussion of labels.

5.3.2 PUBLIC

Syntax: [<label>] PUBLIC <symbol> [<type,...>]

The PUBLIC directive is used to make <symbol> available outside the module in which it is declared.

Example:

```
BSIZE      EQU      29
PUBLIC     BSIZE
```

The above lines cause the value 29 to be available as the symbol "BSIZE" to any other module which is linked together with this module. The Assembler passes the name of the symbol and its value to the Linker so that it can supply them to other modules.

For an explanation of the optional type parameter, see the next section on the EXTERN directive, which also contains an example using the PUBLIC and EXTERN directives.

5.3.3 EXTERN (or EXTRN)

Syntax: [<label>] EXTERN <symbol> [<type,...>]

The EXTERN directive tells the Assembler that the value of <symbol> is not available at assembly-time, but will be supplied later by the Linker.

Example:

```

MODULE ONE
PUBLIC BSIZE EQU 50
.
.
ENDMOD

MODULE TWO
EXTERN BSIZE
LDAA #BSIZE
.
.
END

```

The symbol BSIZE is declared in MODULE ONE. The statement "PUBLIC BSIZE" makes it a global symbol whose name and value are available to the Linker. In MODULE TWO, the statement "EXTERN BSIZE" tells the Assembler that the Linker will supply the value for this symbol (the Linker will get the value from MODULE ONE).

Note that the above two modules may be in separate source files or in the same source file.

The Type Parameter

The PUBLIC and EXTERN directives accept an optional sequence of type numbers in parenthesis following the symbol name. These numbers are used to describe the PUBLICed or EXTERNed symbol. The Assembler includes these numbers in the relocatable output file where they are examined by the XLINK Linker to be sure that they match in every occurrence of the symbol.

This link-time typechecking feature helps confirm that global symbols are used consistently throughout a program, which might consist of many separate modules. For instance, a subroutine entry point in one module should not be referenced as an address of a data byte in another module.

Up to 255 type numbers may be specified for each PUBLIC or EXTERN symbol and each of the numbers may range from 0 through 255, decimal. For programs written only in assembly language, the meaning of these number is completely arbitrary -- it is up to you to determine if and how they will be used (if you do not include any type numbers the Linker will not attempt to typecheck

that symbol). You may use any scheme for assigning numbers that you want, just be sure to use the same numbers for like symbol types.

Consider the following example program fragment consisting of two modules (ONE and TWO) and two global symbols (STATUS and FILL):

"Include" file TYPES.INC:

```

BYTE_DATA EQU 0    type for byte data
WORD_DATA EQU 1    type for word data
ROUTINE EQU 2      type for callable routines
.
. (other types as needed...)

```

Source File #1:

```

MODULE ONE
$TYPES.INC include file with type definitions
EXTERN STATUS(BYTE_DATA) byte data
PUBLIC FILL(ROUTINE) subroutine entry point
.
. (other PUBLIC/EXTERN symbols...)
.
FILL:
LDAA STATUS
.
.

```

Source File #2:

```

MODULE TWO
$TYPES.INC include file with type definitions
PUBLIC STATUS(BYTE_DATA) byte data
EXTERN FILL(ROUTINE) subroutine entry point
.
. (other PUBLIC/EXTERN symbols...)
.
JSR FILL
.
.
STATUS: RMB 1
.

```

The include file TYPES.INC contains definitions (EQUs) for the symbol types you wish to have checked at link-time (BYTE_DATA, WORD_DATA, ROUTINE, etc.). You create as many types as you need for your application using whatever number values you choose

to represent the various types. This file is then included (with the \$ directive) in all modules that will be using typed global symbols.

Note that in modules ONE and TWO we have used the types defined in TYPES.INC as parameters in the PUBLIC and EXTERN directives. The type BYTE_DATA is used for the symbol STATUS in both modules, as this symbol is a label that refers to a reserved data byte in module TWO. The type ROUTINE is used for the symbol FILL, since it is a label that refers to a subroutine entry point in module ONE.

Again, the types we have chosen were selected arbitrarily -- all that matters is that the type for the EXTERN statement matches the type for the PUBLIC statement for a given symbol. The Linker will produce a warning message if the types do not match.

In the above example we specified only a single type number for each global symbol. However, the PUBLIC and EXTERN statements will accept a list of up to 255 numbers. This feature is useful for building complex or compound data types. For example, the following might be used to type a symbol that refers to a function that returns a BYTE_DATA type:

```
EXTERN SOME_FUNCTION(ROUTINE,BYTE_DATA)
```

The Archimedes C-6811 Compiler uses complex data typing for all global symbols if the "-g" compiler option is specified. The type numbers that the Compiler uses are generated by an internal algorithm that is beyond the scope of this discussion. However, there is a simple method of adding typechecking to Assembler modules that will be linked with C code. This is discussed in the C-6811 documentation.

NOTE

If you will be linking Assembler modules with C modules, refer to the C-6811 Compiler chapter for details on adding type information to Assembler symbols.

5.3.4 LOCSYM

Syntax: [`<label>`] `LOCSYM+`
 [`<label>`] `LOCSYM-`

These directives specify that local symbols are to be written to the output object file (.R07 filetype) for later use by a hardware emulator or for inclusion in the Linker cross-reference listing. `LOCSYM+` enables this option and `LOCSYM-` disables it.

Note that this is a "counting" directive, which means that the local symbol option will be turned on if the total number of `LOCSYM+` directives assembled so far is greater than the total number of `LOCSYM-` directives.

The default setting for this directive when you begin a new module is `LOCSYM-`. The Assembler command line `S` option is equivalent to a `LOCSYM+` directive at the beginning of the source code and hence reverses the default setting. (See 2.5.2 Command line Parameters and Options for more information about the `S` option.)

Example:

```

LABEL1:
  LDAA  #1          symbol LABEL1 not output
  LOCSYM+
LABEL2:
  LDAB  #3          symbol LABEL2 is output
  LOCSYM-
LABEL3:
  LDAA  #4          symbol LABEL3 not output
```

5.4 Segment Directives

This section covers the assembler directives which define memory segments.

5.4.1 Overview

A segment is an area of memory in the 68HC11's memory space (either RAM or ROM, internal or external) which extends from a starting location in memory through a contiguous number of bytes without any gaps of undefined memory. Segments are created and named by the user in Assembler programs and located in memory by the Linker.

A segment directive remains in effect until another segment directive is encountered or the module ends. The same segment can be used many times in one or many source files.

Typical 68HC11 Assembler programs consist of at least two relocatable segments: one for executable code, which is linked to the address of your target's ROM; and one for read/write data, which is linked to the address of 68HC11 internal or external memory. However, the quantity, names and uses for the segments are completely up to you. See 3.2.4 Segments for an additional discussion of segments.

NOTE

The C-6811 Compiler creates a set of pre-defined segments that it uses for various types of code and data (CODE, CONST, IDATA, etc.). If you are writing Assembler modules that will be linked with a C program, you must be aware of these segments. For a complete list of segments, see the Compiler section of this manual.

The Program Location Counter

The Assembler associates a separate Program Location Counter with each segment within a given source file. This location counter represents the number of bytes from the beginning of the segment to the current location being assembled. The current value of the Program Location Counter is available for use in expressions through the star symbol (*).

An ORG directive can be used to set the location counter to any value. If an ORG is not used, the Location Counter begins at zero.

Relocatable Code and Data

Please note that, by definition, a segment contains only relocatable code or data -- the final location of instructions or data which follow an RSEG, COMMON or STACK segment directive is determined by the XLINK Linker (see the -Z command in 4.3 XLINK Command Detail in the Linker chapter).

Absolute Code and Data

Absolute code or data (instructions or data which follow an ASEG directive) is, by definition, segmentless. The location of absolute code or data in memory is determined solely by an ORG directive in source code. The Linker does not play any role in determining the location of absolute code or data.

NOTE

We strongly recommend that you use relocatable segments in Assembler programs as much as possible (vs. absolute code and data).

5.4.2 ASEG

Syntax: [<label>] ASEG

The ASEG directive indicates that the source code lines which follow are to be placed at a fixed location in memory, as specified by the ORG directive (described below).

When the ASEG directive is used, the location of the code in the 68HC11's memory is completely determined at the time of assembly and is not dependent upon the Linker.

Example:

```

ASEG
ORG      0
JMP      START    POWER ON VECTOR
ORG      3
JMP      IESRV     EXT INT SERVICE
ORG      $0B
JMP      TOSRV     TIMER INT SERVICE

ORG      $30
START:
LDAA     #0 BEGINNING OF POWER ON CODE
      .
      .

```

In the above example, the reset, the external interrupt and the timer 0 overflow interrupt vectors are assembled at the required locations in 68HC11 memory

5.4.3 RSEG

Syntax: [<label>] RSEG <name> [(<alignment>)]

The RSEG directive declares that the code or data which follows is to be placed in the relocatable segment name (along with any code or data belonging to an RSEG segment of the same name). At link time, the Linker locates all the code or data (from any number of modules) which belong to segment name at the address specified for that segment by the XLINK "-Z" segment location command.

The RSEG directive causes code or data to be allocated from low to high addresses (vs. the STACK directive which causes high-to-low allocation).

Examples:

```

RSEG CODEPART1
LOOP1:
LDAA #1
.
.
RSEG DATAPART1
TAB10:
FCB 10, 20, 30, 40
.
.

```

The code following the first RSEG directive will be placed in the segment named CODEPART1. The data following the second RSEG directive will be placed into the segment named DATAPART1.

Segment Alignment

A relocatable segment may also have an optional alignment operand, enclosed in parentheses. This operand tells the linker that the segment is to start on a specified boundary in memory. Relocation boundaries can be any power of 2: 1, 2, 4, 8, 16, 32, etc. If an alignment operand is not specified, a default alignment of 1 byte (2 to the 0 power) is used. This means that the segment may begin on the next available byte of memory.

When an alignment operand is given, it must evaluate to an integer in the range of zero through 31. The segment is placed beginning at the first available location in memory whose address divides evenly by that power of 2.

Example:

```
RSEG DATA(4) align on a 16-byte boundary (2 to the 4 power)
```

In the above example, the alignment operand of 4 means that the segment DATA can only be placed at the beginning of a paragraph (16 byte) address boundary. In terms of memory addresses this means that if the previous segment ended at address \$123, the Linker will place the segment DATA at address \$130 (the beginning of the next paragraph).

5.4.4 COMMON

Syntax: [<label>] COMMON <name> [(<alignment>)]

The COMMON directive declares that the code or data which follows is to be placed in the relocatable segment name. A COMMON-type segment will be located in memory at the same location as other COMMON segments that have the same name. In other words, all COMMON segments of the same name will start at the same location in memory and overlay each other.

Obviously, the COMMON segment type should not be used for executable code. A typical application would be where you desire to have a number of different routines share a reusable, common area of memory for data.

The final size of the COMMON segment name in memory is determined by the size of largest occurrence of this segment. The location in memory is determined by the Linker (see the "-Z" command in 4.3 XLINK Command Detail section in the Linker chapter).

Example:

```

File 1:
COMMON DATA1
FLAG1:  RMB    1
.
COMMON DATA2
MSG1:   RMB    1
.

File 2:
COMMON DATA1
FLAG2:  RMB    1
.
COMMON DATA2
MSG2:   RMB    1
.

```

The above data allocations will be placed in two relocatable, COMMON-type segments called DATA1 and DATA2. The two labels FLAG1 and FLAG2 both refer to the same location in memory at the beginning of the COMMON segment DATA1. The labels MSG1 and MSG2 both refer to the same location at the beginning of the COMMON segment DATA2.

See Segment Alignment under 5.4.3 RSEG for information on using the alignment parameter.

5.4.5 STACK

Syntax: [<label>] STACK <name>[(<alignment>)]

The STACK directive declares that the code or data which follows is to be placed in the segment name (along with any code or data belonging to a STACK segment of the same name). At link time, the Linker locates all the code or data (from any number of modules) which belong to segment name at the address specified for that segment by the XLINK "-Z" segment location command.

The STACK directive causes code or data to be allocated from high to low addresses (vs. the RSEG directive which causes low-to-high allocation).

Example:

```

STACK  PARSESTACK
RMB    512
.

```

The code following the STACK directive above will be placed into a relocatable STACK segment called PARSESTACK.

See Segment Alignment under 5.4.3 RSEG for information on using the alignment parameter.

5.4.6 ORG

Syntax: [<label>] ORG <expression>

The ORG directive allows you to set the value of the Program Location Counter anywhere within a segment, or at an absolute value if you are assembling absolute code or data following an ASEG directive (see 5.4.2 ASEG).

ORG in Relocatable Segments

In a relocatable segment (following an RSEG, COMMON or STACK directive), the ORG statement accepts only a relocatable expression as a parameter. This expression can be specified in the form of:

```
ORG      * + nn
```

```
ORG      label + nn
```

where "nn" bytes are added to the current value of the Program Location Counter (*) or to a relocatable address (label).

ORG may NOT make any forward reference to symbols which are defined in the source code module.

Example of ORG used in a relocatable segment:

```
RSEG      CODE
ORG      * + $40    skip 40h bytes of code at the beginning
.
.
.
LABEL1:
JMP      SOMEWHERE
.
ORG      LABEL1 + $20    start 20h after LABEL1
LDAA     #1
.
.
.
```

In the above example, the first ORG statement leaves a "hole" in the beginning of the segment CODE. The second ORG statement sets the location counter at 20h bytes after the (relocatable) label LABEL1. This feature can be used to leave room in a segment for interrupt vectors or other items that have a fixed address in memory.

ORG in Absolute Code or Data

When it comes after an ASEG directive, an ORG statement sets the absolute (final) location in memory of the code or data that follows it in the program.

Example of ORG in absolute code:

```
ASEG
ORG      $8000
ENTRY2:
LDAA     #1 this code will be fixed at addr $8000
.
```

WARNING!

Use caution when specifying the location of absolute code or data so that you do not "ORG" one object at the same address as another object, because the Linker will NOT warn you of such a condition!

5.5 Value Directives

This section covers the Assembler Directives which are used to assign values to symbols: EQU, SET and DEFINE.

5.5.1 Overview

Value directives allow you to create a symbol and assign it a particular constant numeric value. You may then use the symbol in your source code in place of a constant number.

This is most useful for numbers which may possibly change in later revisions of your program. Changing the value assigned to a symbol

causes ALL the code using that symbol to use the new value without having to edit all of the individual statements in which it is used.

5.5.2 EQU (or =)

Syntax: <symbol> EQU <expression>
 <symbol> = <expression>

The EQU directive (or = for short) creates a symbol named symbol and assigns to it the value of expression. The symbol and expression parts of the EQU statement must be provided.

This symbol is valid only in the module in which the EQU statement appears (also see the DEFINE directive, below, which is similar to EQU, but makes the symbol available throughout the entire source file in which it appears).

A symbol which has been given a value with an EQU directive can be made available to other modules with the use of the PUBLIC directive.

Please note that a symbol which has been given a value with an EQU directive may NOT be redefined within the same module (also see the SET directive below, which allows symbols to be redefined within the same module).

Example:

```
BUFSIZE    EQU    100
KEY         =     25
```

The first statement sets the symbol BUFSIZE equal to 100. The second statement sets the label KEY equal to 25. The following statements can use these labels instead of the numbers they represent:

```
LDAA    #BUFSIZE
LDAB    #(BUFSIZE / 10)
```

5.5.3 SET

Syntax: <symbol> SET <expression>

The SET directive creates a symbol named symbol and assigns to it the value of expression. The symbol and expression parts of the SET statement must be provided.

Unlike the EQU directive, symbols defined with SET can be redefined within a module.

A symbol defined with SET is only available within the module where it appears and may NOT be made available to other modules with the PUBLIC directive.

Example:

```
STARTAT      SET      3
LDAA         #STARTAT

STARTAT      SET      5
LDAB         #STARTAT
```

The above statements will cause accumulator A to be loaded with the number 3 and accumulator B to be loaded with the number 5.

5.5.4 DEFINE

Syntax: <symbol> DEFINE <expression>

The DEFINE directive creates a symbol named symbol and assigns to it the value of expression. The symbol and expression parts of the DEFINE statement must be provided.

This symbol is valid throughout the entire source file in which the DEFINE statement appears (also see the EQU directive, above, in which the symbol is only available in the current module).

A symbol which has been given a value with a DEFINE directive can be made available to other modules with the use of the PUBLIC directive.

Please note that a symbol which has been given a value with an DEFINE directive may NOT be redefined within the same module (also see the SET directive above, which allows symbols to be redefined).

NOTE

The DEFINE directive makes a value available to all the modules within a given source file AT ASSEMBLY TIME. The PUBLIC directive makes a value available to all modules within a program AT LINK TIME.

Example:

```

NAME      ONE
RSEG      CODE
VALUE1: DEFINE $32
LDAA      #1
.
.
ENDMOD

NAME      TWO
RSEG      DATA
FCB        VALUE1
.
.
END

```

Notice that even though symbol VALUE1 is defined in module ONE it is still accessible in module TWO. VALUE1 would not, however, be accessible to modules outside of this source file (unless it is declared as PUBLIC).

5.6 Data Directives

This section covers the Assembler Data Directives which allow you to reserve and initialize data memory: FCB, FCC, FDB, FQB, and RMB.

5.6.1 Overview

The Assembler data directives are used to declare locations in 68HC11 memory to be used for data storage. The RMB directive is used to reserve space for read/write data elements (i.e. variables). The FCB, FCC, FDB, and FQB directives are used to both reserve space in memory and to initialize that memory with fixed data values.

In typical 68HC11 use, uninitialized data storage (defined with the RMB directive) is in read/write memory, since the contents of RAM is not guaranteed after power-up of the chip. Initialized data storage (defined with the FCB, FDB, FCC, and FQB directives) is typically in read-only memory since initialized data can be permanently programmed into ROM.

When using hardware emulators, be careful to consider the issue of RAM versus ROM, and initialized data versus uninitialized data.

5.6.2 RMB

Syntax: [<label>] RMB <expression>

The RMB directive ("Reserve Memory Byte") causes a given number of bytes, specified by expression, to be reserved at the current location in the program. In other words, the Program Location Counter for the current segment is incremented by the number of bytes specified in expression.

Example:

```
MSGBUF      RMB      80
DIRECTORY   RMB      RECORD_SIZE * NUM_PEOPLE
```

The first line reserves 80 bytes of storage for a buffer, with the label MSGBUF equal to the beginning address of the buffer. In the second line, please note that the values of the two symbols in the expression MUST be available at assembly-time (i.e., they cannot be defined as EXTERN).

5.6.3 FCB

Syntax: [<label>] FCB <expression,...>
 [<label>] FCB '<ASCII string>','...

The FCB ("Fill Constant Byte") directive causes memory locations to be reserved AND initialized to the values listed in the expression or ASCII string parameters. The number of bytes reserved depends upon the number of parameters given. Multiple expression or ASCII string parameters can be entered on one line by separating them with commas (,), and numeric expressions and ASCII strings can be mixed together.

The expression parameters can consist of any valid absolute, relocatable or external expression that evaluates to a number in the range of -128 to 255, decimal.

An ASCII string parameter must be enclosed in single quotes ('). Any printable character or a space (values from 32 to 126, decimal) can be used in the string. Multiple strings can be specified on one line by separating them with a comma (,). Each character in the string will take up one byte of the reserved space.

Examples:

```

1 0000
2 0000
3
4 0000 07          CONST: FCB      111B
5 0001 8000FF      FCB      -128,0,255
6 0004 54686520    MSGX: FCB      'The rain in Spain', $0D, $0A
   0008 7261696E
   000C 20696E20
   0010 53706169
   0014 6E0D0A
7 0017 05          FCB      SYM1 + 5
8
9 0018             END

```

The above Assembler listing illustrates the use of the FCB directive to create initialized data. Please note that the segment CONSTANTS used for this data should be linked to an address in ROM. Line #7 contains a FCB directive that includes an external expression that will be resolved by the Linker.

See further examples in section 5.6.5 FQB.

5.6.4 FDB

```

Syntax:  [<label>] FDB <expression,...>
         [<label>] FDB '<ASCII constant>',...

```

The FDB ("Fill Double Byte") directive causes words (2 bytes/16 bits) of memory to be reserved AND initialized to the values listed in the expression or ASCII constant parameters. The number of words reserved is equal to the number of parameters specified. Multiple expression or ASCII constant parameters can be entered on one line by separating the parameters with commas (,). Numeric expressions and ASCII constants can be mixed on one line.

The expression parameters can consist of any valid absolute, relocatable or external expression that evaluates to a number in the range of -32,768 to 65,535.

An ASCII constant parameter can consist of one or two characters, which must be enclosed in single quotes (''). Any printable character or a space (values from 32 to 126, decimal) can be used. Multiple ASCII constants can be entered on one line by separating them with commas (,). Each constant will take up one word (2 bytes) in memory (a single character still causes two bytes to be reserved).

For a numeric expression, the first byte in memory (i.e., the lower memory address) will be initialized with the most significant byte of the word.

For single ASCII characters ('A'), the higher memory address will be initialized with the character's ASCII code; a zero is placed in the lower address in memory. For a pair of ASCII characters ('AB'), the first character of the pair ('A') goes into the lower memory address and the second character ('B') into the higher memory address.

5.6.5 FQB

Syntax: [<label>] FQB <expression,...>
 [<label>] FQB '<ASCII constant>','...

The FQB ("Fill Quad Byte") directive causes double words (4 bytes/32 bits) of memory to be reserved AND initialized to the values listed in the expression or ASCII constant parameters. The number of double words reserved is equal to the number of parameters specified. Multiple expression or ASCII constant parameters can be entered on one line by separating the parameters with commas (,). Numeric expressions and ASCII constants can be mixed on one line.

The expression parameters can consist of any valid absolute, relocatable or external expression that evaluates to a number in the range of -2,147,483,648 to 4,294,967,295. Also, the FQB directive accepts real (floating point) number constants expressed in "scientific notation" as data initializers (see the examples, below).

If an integer expression is specified as a parameter, the first byte (lowest address) in initialized memory will contain the most significant byte of the double word. The last byte (highest address) in memory will contain the least significant byte of the double word. The following diagram further illustrates how the double-word constant "\$29EA0174" would be stored in memory beginning at location \$82:

Address:	82	83	84	85
Content:	29	EA	01	74

Real (floating point) number initializers are placed in memory in the IEEE single precision real number format.

An ASCII constant parameter can consist of 1 to 4 characters, which must be enclosed in single quotes (''). Any printable character or a space (values from 32 to 126, decimal) can be used. Multiple sets of characters can be specified on one line by separating each set with a comma (,). Each character or set of characters will take up a double word (4 bytes) in memory (a single character will still cause 4 bytes to be reserved).

If an ASCII constant is specified as a parameter (e.g. 'ABCD'), the first byte (lowest address) in initialized memory will contain the first character ('A') of the double word. The last byte (highest address) in memory will contain the last character ('D') in the double word. If fewer than 4 characters are specified, the unused memory bytes are initialized to zero. The diagram below shows how the double word ASCII constant 'PQRS' is stored in memory:

Address:	82	83	84	85
Content:	'P'	'Q'	'R'	'S'

Examples:

The following listing shows comparisons of data directive results after assembly. The second column shows memory addresses and the third column shows memory contents:

2	0000	RMB	10	* uninitialized
3	000A 7B	FCB	123	* initialized as bytes
4	000B 007B	FDB	123	* ...words and
5	000D 0000007B	FQB	123	* ...double words
6	0011 85	FCB	-123	* negative integers
7	0012 FF85	FDB	-123	
8	0014 FFFFFFF85	FQB	-123	
9	0018 3039	FDB	12345	* large integers
10	001A 00003039	FQB	12345	
11	001E 00BC614E	FQB	12345678	
12	0022 12345678	FQB	\$12345678	
13	0026 41	FCB	'A'	* ASCII characters
14	0027 0041	FDB	'A'	
15	0029 00000041	FQB	'A'	
16	002D 4142	FCB	'AB'	* ASCII pairs
17	002F 4142	FDB	'AB'	
18	0031 00004142	FQB	'AB'	
19	0035 41424344	FCB	'ABCD'	* ASCII quads
20	0039 41424344	FQB	'ABCD'	
21	003D 3F800000	FQB	"1	* real numbers -
22	0041 BF800000	FQB	"-1	* IEEE format
23	0045 3FC00000	FQB	"1.5	

5.6.6 FCC

Syntax: [<label>] FCC <'string'>

The FCC (Fil Constant Character) directive initializes memory with the ASCII string specified in the argument.

Note that the Motorola syntax variation "expr,string" is not supported. Single quotes are the only allowed string delimiters.

5.6.7 ZPAGE

Syntax: EXTERN label:ZPAGE
 RSEG <segment name>
 ZPAGE
 label1 RMB 1
 label2 RMB 1
 LABEL3 EQU \$1000

When the optional :ZPAGE directive is added to an EXTERNAL symbol, the assembler will be forced to use the direct address mode when referenced.

By using the ZPAGE directive, labels defined by the current relocatable segment will be accessed using the direct access mode (short addresses). Only label1 and label2 are accessed using short addressing mode, because LABEL3 is equated to an absolute value.

Note, however, that forward referencing is not allowed when using the ZPAGE directive. Therefore, all ZPAGE symbols must be defined before they are referenced.

Example:

```
RSEG SHORTAD
ZPAGE
C:  RMB 1
    RSEG CODE
    LDAB #12
    STAB
```

In this example, the opcode generated for the STAB instruction is D700 instead of F70000.

5.7 Addressing Modes

This section briefly describes, how the addressing modes of the 68HC11 should be expressed in assembly language. Note that range checking is performed by A6801 as well as by XLINK.

Implied (no operands):

CLRA TB etc...

Immediate:

#Expr

In the immediate mode the expression can be either absolute, relocatable (including segment functions), or external. Range is limited to -128 to 255 for 8-bit operands (LDAA #Expr) while 16-bit operands (LDX #Expr) have a range of -32768 to 65535.

Direct:

Expr<Expr

In the direct mode the expression can be either absolute, relocatable (including segment functions), or external. The first form will make the assembler select the direct mode, if the expression is absolute and contains no forward references and is within reach for the direct mode, or in the case of an external or relocatable symbol, if the symbol belongs to the zero page (i.e. is defined by the ZPAGE directive). By using the form "<Expr", the assembler can be *forced* to use the direct mode. Range is limited to 0 to 255.

Extended:

Expr>Expr

In the extended mode the expression can be either absolute, relocatable (including segment functions) or external. The first form will make the assembler select the extended mode, if the expression contains forward references, or is external or relocatable (and not belongs to ZPAGE), or if the expression is absolute and outside range for the direct mode. By using the form ">Expr", the assembler can be *forced* to use the extended mode. Range is limited to 0 to 65535.

Indexed (8-bit offset):

X ,X Expr,X

In the indexed mode the expression can be either absolute, relocatable (including segment functions) or external. Range is limited to 0 to 255.

Relative:

Expr

Branch instructions are PC relative (-126 to +129) and are only allowed to refer to addresses in the current segment.

Immediate to direct (6301):

#Expr1, Expr2

The immediate value (Expr1) and the direct address (Expr2) follow the same rules as other 8-bit immediate values and direct addresses respectively.

Immediate to indexed (6301):

#Expr1, X #Expr1, Expr2, X

The immediate value (Expr1) and the offset (Expr2) follow the same rules as other 8-bit immediate values and indexed addresses respectively. In the first example offset is assumed to be zero.

5.7 Conditional Assembly Directives

This section covers the conditional assembly directives which provide logical control over selective assembly of source code: IF, ENDIF and ELSE.

5.7.1 Overview of Conditional Assembly

The conditional assembly directives give you assembly-time control over which sections of a given source file will be assembled when building a program. This has the advantage of allowing a single source code file to be used for multiple purposes or with multiple configuration options. The configuration of the assembly is controlled by one or more values which are defined at assembly-time within the program (usually by an EQU, DEFINE or SET data directive).

A very typical use for conditional assembly is to temporarily include (or possibly block out) a section of code in the program for debugging purposes, without having to actually modify source lines. This is done by surrounding the "debug" code with an "IF FCBUG ... ENDIF" pair:

```

DEBUG      EQU      1          set to 1 if debug, 0 if production
.
.
IF DEBUG    **** DEBUG ****
TEST3:      LDAA     #ERRORS
            JSR      PRINTIT.
.
. (test code that you wish to include in the assembly
.  only during debugging, but not in production)
.
ENDIF      **** DEBUG ****

```

By simply setting the "DEBUG" equate to 1, the TEST3 routine will be assembled into the program. This routine could print out the value of a variable you wish to monitor (e.g. ERRORS) during the debugging process. For the "production" version of the program you would set the FCBUG equate to "0" to eliminate this code in subsequent assemblies.

5.7.2 IF

Syntax: [<label>] IF <condition>

The IF directive is placed at the beginning of a portion of source code which is to be assembled or not assembled depending upon the value of <condition>. If <condition> evaluates to FALSE at the time the IF directive is assembled, the source code lines following the IF directive, up to the next ENDIF directive, are not included in the assembly. If <condition> evaluates to TRUE, then the lines are assembled.

An IF <condition> can consist of any valid expression that can be fully evaluated at assembly time (expressions involving external symbols cannot be used). If a numeric expression evaluates to zero then it is considered FALSE. If it evaluates to anything other than zero (including a negative value), it is considered TRUE.

An expression involving an equality or comparison operator simply evaluates to TRUE or FALSE. In the following example, the expression (VERSION > 10) is TRUE since the symbol VERSION has been set to 11. The routine NEW_CODE will be assembled:

```
VERSION      EQU      11
.
IF (VERSION > 10)
NEW_CODE:
.
.
ENDIF
```

Please see section 4: Assembler Expression Reference for more information on forming and using expressions.

Please note that IF-ENDIF blocks can be nested up to any depth. This means that an entire IF-ENDIF block of code may be contained within another IF-ENDIF block, and so on.

5.7.3 ENDIF

Syntax: [<label>] ENDIF

The ENDIF directive marks the end of a block of "conditionally assembled" code which began with an IF directive. IF and ENDIF must always be used in matching pairs.

5.7.4 ELSE

Syntax: [<label>] ELSE

The ELSE directive allows you to split up a conditionally-assembled section of code into two parts: one section which will be assembled and one which will not be assembled, depending on the IF condition.

The code from the IF directive to the next ELSE directive will be assembled only when the IF condition is TRUE. The code from the ELSE directive to the ENDIF directive will only be assembled when the IF condition is FALSE.

Example:

```

VERSION      EQU      11
.
.
NEW_CODE:    IF      (VERSION > 10)
.
.
.
OLD_CODE:    ELSE      the version number must be < 11
.
.
.
ENDIF

```

If VERSION is greater than 10, the code following NEW_CODE will be assembled, otherwise OLD_CODE will be assembled.

Conditional Assembly Example

The following is a listing showing the effects of an example conditional assembly:

```

3          IF      0          FALSE expression
4          LABEL1: LDAA  #1
5 0000      ELSE
6 0000 7A02 LABEL2: LDAB  #2
7 0002      ENDIF
8
9 0002      IF      (2 + 2) = 4  TRUE expression
10 0002 7B03 LABEL3: LDAA  #3
11          ELSE
12          LABEL4: LDAB  #4
13 0004      ENDIF
14
15          IF      (2 + 2) <> 4  FALSE expression
16          LABEL5: LDAB  #5
17 0004      ELSE
18 0004 7E06 LABEL6: LDAA  #6
19 0006      ENDIF
20 0006      END

```

5.8 Source Code Expansion Directives

This section describes the include (\$) assembler directive and refers to section 7: Assembler Macros for a description of the macro directives.

5.8.1 Overview

The normal course of assembler operation is for it to read one source code line after another from the source file and process each line in turn. There are two ways in which this normal sequence is altered:

- An "include file" may be specified with the "\$" directive, in which case the Assembler temporarily switches to reading source lines from that file until it is exhausted, after which it returns to the file it was previously reading.
- An inline "macro" can be invoked in the source code, in which case the "macro processor" portion of the Assembler generates the source code lines and feeds them one after another to be assembled. When the macro terminates, the Assembler returns to reading the source file directly again. The macro directives MACRO, ENDMAC and % are used to define and invoke assembler macros.

NOTE

The Macro directives are not described here but in section 6: Assembler Macros. Please refer to this section for a full description of macro operation.

5.8.2 \$ (Include file)

Syntax: \$<include file>

The \$ directive, in effect, causes the Assembler to insert another source file with the name include file into the stream of source lines it is currently reading. The Assembler temporarily stops assembling the current source file and assembles lines from the include file instead. When the end of the include file is reached, it returns to reading the previous file.

The \$ character MUST be located in the first column of the source line. The include file name MUST begin immediately after the \$, in the second column (see the example below).

If the include file name does not contain a filetype, the default filetype of ".S07" will be used. However, a filetype of ".INC" is recommended for include files to differentiate them from normal source files.

Include files may be nested three deep. The original source file can include another file, which can include a second file, which can include a third file.

MS-DOS users, please note that if you intend to nest include files, your CONFIG.SYS file (located in the root directory of the boot drive) must have a FILES statement that allows for at least 20 open files (e.g., FILES=20).

Include files are often used for common data definitions (EQUs, SETs and DEFINEs) that will be needed in a number of different source files. Using an include file eliminates the need to enter these definitions in each individual source file:

```
* local definitions
VALUE1      EQU      1
VALUE2      EQU      2

* common definitions
$DATADEFS.INC      include file with common data defs
.
.
```

5.9 Listing Directives

This section covers the directives which allow you to control Assembler listings: LSTOUT, LSTCND, LSTCOD, LSTMAC, LSTEXP, LSTWID, LSTFOR, LSTPAG, PAGESIZ, PAGE, TITL, STITL, PSTITL and LSTXRF.

5.9.1 Overview

Listing directives allow you to vary the form and content of listings produced by the Assembler. The things you can control include:

- Which parts of your assembly language source code are included in the listing
- How each line on the listing is formatted
- How the pages of your listing are formatted

Whether or not a listing is generated, and the destination of that listing (file or device name), is determined when you run the Assembler. The listing directives simply control the content and format of the list output. A listing is generated when you specify a

file or device name as the second parameter on the A6801 command line:

```
A6801 MONITOR MONITOR
```

```
A6801 MONITOR LPT1:
```

The first example causes a list file named MONITOR.LST to be produced; the second line sends the listing directly to the MS-DOS LPT1: device (the parallel printer connected to port LPT1:).

Counting (+/-) Directives

Please note that several of the listing directives (those which end with a plus or minus sign) are referred to as "counting" directives. A counting directive will take effect only if the number of "+" directives is greater than the number of "-" directives encountered up to that point in the program.

5.9.2 LSTOUT

Syntax: [<label>] LSTOUT+
 [<label>] LSTOUT-

The LSTOUT+ and LSTOUT- directives cause the listing process to be suspended (LSTOUT-) or resumed (LSTOUT+) at the point where the directive occurs. If a LSTOUT directive is not specified, the default setting of LSTOUT+ applies.

Given the following source code ...

```
LABEL0:    LDAA    #0  
          LSTOUT-  
LABEL1:    LDAB    #1  
          LSTOUT+  
LABEL2:    LDAA    #2
```

... the following lines would be placed into the listing:

```
2 0000 7800    LABEL0: LDAA    #0  
6 0004 7A02    LABEL2: LDAA    #2
```


5.9.3 LSTCND

Syntax: [<label>] LSTCND+
 [<label>] LSTCND-

The LSTCND+ and LSTCND- directives cause the listing to include (LSTCND-) or exclude (LSTCND+) source lines which are not assembled because of a conditional assembly directive (IF-ENDIF). If a LSTCND directive is not specified, a default of LSTCND- is used (i.e., all source lines are included in the listing even if they are not assembled).

Given the following source code ...

```
TRUEQU      1
  LSTCND+
  IF      TRUE
LABEL0:     LDAA      #0
  ELSE
LABEL1:     LDAB      #1
  ENDIF

  LSTCND-
  IF      TRUE
LABEL2:     LDAA      #2
  ELSE
LABEL3:     LDAB      #3
  ENDIF
LABEL4:     LDAB      #4
```

... the following lines would be placed in the listing:

```
2 0001          TRUE      EQU      1
3 0000          LSTCND+
4 0000          IF      TRUE
5 0000 7800     LABEL0: LDAA      #0
9
10 0002          LSTCND-
11 0002          IF      TRUE
12 0002 7A02     LABEL2: LDAA      #2
13
14              LABEL3: LDAB      #3
15              ENDIF
16 0004 7C04     LABEL4: LDAB      #4
```

5.9.4 LSTCOD

Syntax: [<label>] STCOD+
 [<label>] LSTCOD-

The LSTCOD+ and LSTCOD- directives used to exclude (LSTCOD-) or include (LSTCOD+) extra lines in the listing that show all of the bytes generated by a long data directive such as a FCB with and ASCII string following it. If a LSTCOD directive is not specified, the default of LSTCOD+ takes effect.

Given the following source code ...

```

LSTCOD-
LABEL1:      FCB      'ABCDEFGH IJKLMNOPQRSTU V'

LSTCOD+
LABEL2:      FCB      'ABCDEFGH IJKLMNOPQRSTU V'

LABEL3:
```

... the following lines would be placed into the listing file:

```

1  000C                                LSTCOD-
2  000C 41424344 LABEL1: FCB      'ABCDEFGH IJKLMNOPQRSTU V'
3
4  0026                                LSTCOD+
5  0026 41424344 LABEL2: FCB      'ABCDEFGH IJKLMNOPQRSTU V'
   002A 45464748
   002E 494A4B4C
   0032 4D4E4F50
   0036 51525354
   003A 5556
6
7  003C                                LABEL3:
```

5.9.5 LSTMAC

Syntax: [<label>] LSTMAC+
 [<label>] LSTMAC-

The LSTMAC+ and LSTMAC- directives are used to exclude (LSTMAC-) or include (LSTMAC+) source lines which define macros. The LSTMAC+ or LSTMAC- directive statements themselves will not appear in the listing. If a LSTMAC directive is not specified, the default of LSTMAC+ takes effect.

Given the following source code ...

```

LABEL1:
  LSTMAC-
  MACRO    %BIGCALL
  LDD      #\0
  JSR      \1
  ENDMAC
LABEL2:
  LSTMAC+
  MACRO    %SMALLCALL
  LDD      #0
  JSR      \0
  ENDMAC
LABEL3:

```

... the following lines would be placed into the listing file:

```

1 0000      LABEL1:
7 0000      LABEL2:
9 0000
10
11          MACRO    %SMALLCALL
12          LDD      #0
13          JSR      \0
          ENDMAC
          LABEL3:

```

Also see section 6: Assembler Macros, for more information on the use of the macro facility.

5.9.6 LSTEXP

Syntax: [<label>] LSTEXP+
 [<label>] LSTEXP-

The LSTEXP+ and LSTEXP- directives are used to exclude (LSTEXP-) or include (LSTEXP+) lines generated by macro expansion. The LSTEXP+ or LSTEXP- directive statements themselves will not appear in the listing. If a LSTEXP directive is not specified, the default of LSTEXP+ takes effect.

Given the following source code ...

```

LABEL1:
  LSTEXP-
  %BIGCALL 10,LABEL8
LABEL2:
  LSTEXP+
  %BIGCALL 20,LABEL9
LABEL3:

```

... the following lines would be placed into the listing file:

```

1 0000          LABEL1:
3 0000          %BIGCALL 10,LABEL8
4          LABEL2:
6 0006          %BIGCALL 20,LABEL9
/ 0006 900014    LDD      #20
/ 0009 120026    JSR      LABEL9
7          LABEL3:

```

5.9.7 LSTWID

Syntax: [<label>] LSTWID+
 [<label>] LSTWID-

The LSTWID+ and LSTWID- directives are used to exclude (LSTWID-) or include (LSTWID+) two extra numbers on each listing line to show the current nesting level of include files and the current line number within each include file.

If a LSTWID directive is not specified, the default of LSTWID- takes effect. Please note that the default for this directive will be changed to LSTWID+ if the W Assembler command line option is specified (see 2.5.2 Command line Parameters and Options).

Given the following source code ...

```

LABEL1:
  LSTWID+
  LDAA      #100
$INCFIL
  LDAB      #101
LABEL2:

  LSTWID-
  LDAA      #200
$INCFIL
  LDAA      #201
LABEL3:

```

... and an include file INCFILE.S07 containing three lines:

```

LDAA      #11
LDAA      #12
LDAA      #13

```

... the following lines would be placed into the listing file:

```

1 0000 LABEL1:
* 2 0 2 0000 LSTWID+
* 3 0 3 0000 7864 LDAA #100
* 4 0 4 $INCFIL
* 5 1 1 0002 780B LDAA #11
* 6 1 2 0004 780C LDAA #12
* 7 1 3 0006 780D LDAA #13
* 8 0 5 0008 7865 LDAB #101
* 9 0 6 LABEL2:
10 0 7
11 000A LSTWID-
12 000A 78C8 LDAA #200
13 $INCFIL
14 000C 780B LDAA #11
15 000E 780C LDAA #12
16 0010 780D LDAA #13
17 0012 78C9 LDAA #201
18 0012 LABEL3:

```

The lines that are in the "wide" listing format are marked above with an "*" (the "*" does not appear in the actual listing). The above example wide listing includes the following elements:

- Listing file line number (2-9)
- Include file nesting depth (0 means lines are from original source file, 1 means they are from include file INCFILE.S07)
- Source file line number (2-4 from original file, 1-3 from the include file, then 5-7 from the original file again)
- Code address
- Label
- Opcode
- Operands

The elements of the normal, narrow listing lines (those not marked with an "*") are:

- Listing file line number (1, 10-18)
- Code address
- Label
- Opcode
- Operands

5.9.8 LSTFOR

Syntax: [<label>] LSTFOR+
 [<label>] LSTFOR-

The LSTFOR+ and LSTFOR- directives are used to enable (LSTFOR+) the "formatted" option for source lines in the listing, or to leave the source lines just as they appear in the original source file (LSTFOR-).

The "formatted" option causes source lines to be formatted with consistent, fixed field positions for a better appearance in the listing. When the LSTFOR- directive is used, the only formatting that will occur is that tab characters are expanded.

If a LSTFOR directive is not specified, the default of LSTFOR- takes effect. Please note that the default for this directive will be changed to LSTFOR+ if the F Assembler command line option is specified (see 2.5.2 Command line Parameters and Options).

Given the following (unformatted) source code ...

```

LABEL1:
  LSTFOR+
  LDAA  #130
  LDAB  #131
LABEL2:
  LSTFOR-
  LDAA  #140
  LDAB  #141
LABEL3:
  
```

... the following lines would be placed into the listing file:

```

1 0000          LABEL1:
2 0000          LSTFOR  +
3 0000 7882     LDAA   #130
4 0002 7883     LDAB   #131
5
6 0004          LABEL2:
7 0004 788C     LSTFOR-
8 0006 788D     LDAA   #140
9              LDAB   #141
          LABEL3:
  
```

5.9.9 LSTPAG+, LSTPAG-

Syntax: [<label>] LSTPAG+
 [<label>] LSTPAG-

The LSTPAG+ and LSTPAG- directives are used to format pages in the listing with page breaks (LSTPAG+), or to leave the page formatting up to the source file contents (LSTPAG-). This directive is used in conjunction with the PAGE, PAGESIZE, and PTITL directives.

If a LSTPAG directive is not specified, the default of LSTPAG- takes effect. Please note that the default for this directive will be changed to LSTPAG+ if the P Assembler command line option is specified (see 2.5.2 Command line Parameters and Options).

5.9.10 PAGESIZ

Syntax: [<label>] PAGESIZ <expression>

The PAGESIZ directive is used to specify the number of lines per page for the listing file that will be used if the LSTPAG+ (page formatting) directive is in effect.

The expression is the number of lines per page, and can range from 10 to 150 lines. If a PAGESIZ directive is not specified, a default of 44 lines/page will take effect. Please note that this default can be changed with the P=nn Assembler command line option (see 2.5.2 Command line Parameters and Options).

5.9.11 PAGE

Syntax: [<label>] PAGE

The PAGE directive causes an immediate page break, regardless of the number of lines already on the current page of the listing. The PAGE directive itself will not appear in the listing. For an example, see the STITL directive, below.

5.9.12 TITL

Syntax: [<label>] TITL '<string>'

The TITL (Title) directive is used to specify a title (string) for the listing that will be placed at the top of each page. After this directive is assembled, anytime a page break occurs, string will be placed at the top of the page. A title is not included in the listing if this directive is not used.

Example:

```
TITL      'Adams Automation Project'
```

5.9.13 PTITL

Syntax: [<label>] PTITL '<string>'

The PTITL (Page/Title) directive is a combination of the PAGE and TITL directives. It causes a page break AND specifies a string to be used as the title for subsequent listing pages (superceding any existing TITL directive).

5.9.14 STITL

Syntax: [<label>] STITL '<string>'

The STITL (Sub-header Title) directive is similar to the TITL directive but it specifies a string to be used for a second title line on the top of each page. This allows you to format the listing with a sub-title for each sub-section of code, while maintaining an overall title at the top of the listing.

For example, if a source file contained the following statements...

```
NAME      TITLE_TEST
LSTPAG+
TITL      'Main Title'
STITL     'Sub Heading Title'
PAGE
END
```

... the listing would appear as below:

2	0000	NAME	TITLE_TEST
3	0000	LSTPAG+	
4	0000	TITL	'Main Title'
5	0000	STITL	'Sub Heading Title'

```
----- ( top of the next page ) -----  
Main Title                               Page 2  
Sub Heading Title  
  
7 0000                                END
```

5.9.15 PSTITL

Syntax: [<label>] PSTITL '<string>'

The PSTITL directive is a combination of the PAGE directive and the STITL directive. It causes an immediate page break AND specifies a new sub-title (string) for subsequent pages.

5.9.16 LSTXRF

Syntax: [<label>] LSTXRF

The LSTXRF directive specifies that a cross reference table is to be generated at the end of the Assembler listing. Specifying this directive is the equivalent of using the X command line option (see 2.5.2 Command line Parameters and Options).

The cross reference table includes a list of symbolic items in your program showing on which lines they are declared and referenced. It lists symbols (public, external and local), segments and macros.

Each symbolic item in the "Symbol and Cross Reference Table" includes:

- The Symbol name
- The symbol's Value (or length, if it is a segment)
- The symbol's Type
- The source line number where the symbol is defined (Define)
- A list of all source line numbers where the symbol is referenced (Refline)

In the cross reference table, macros have neither value nor type.

The Type field in the Symbol Cross Reference for non-macro symbols is coded follows:

A Absolute, local symbol
 A* Absolute, local symbol from a DEFINE directive
 AE Absolute, public symbol
 AE* Absolute, public symbol from a DEFINE directive
 V Variable symbol from a SET directive
 U Undefined symbol
 SnnR RSEG type reloc segment (with ID # nn)
 SnnS STACK type reloc segment (with ID # nn)
 SnnC COMMON type reloc segment (with ID # nn)
 Xnn Symbol from an EXTERN directive (with ID # nn)
 Rnn Local relocatable symbol in segment nn
 RnnE Public relocatable symbol in segment nn

A typical cross-reference listing is shown below:

```

1 0000          LSTXRF
2 0000          NAME DEMOXREF
3 0000          RSEG CODE_1
4 0019          VALUE1: EQU 25
5 001A          VALUE2: EQU 26
6 0000          PUBLIC VALUE2
7 001B          VALUE3: DEFINE 27
8 001C          VALUE4: DEFINE 28
9 0000          PUBLIC VALUE4
10 001D         VALUE5: SET 29
11 0000         EXTERN VALUE6
12 0000 8601     LABEL1: LDAA #1
13 0000         STACK OPER_STACK
14 0000 C602     LABEL2: LDAB #2
15 0002         PUBLIC LABEL2
16 0000         COMMON SHARE_DATA
17 0000 4D455353 LABEL3: FCC 'MESSAGE TEXT'
    0004 41474520
    0008 54455854
18 0000         RSEG CODE_2
19 0000 8604     LABEL4: LDAA #4
20 0002         MACRO %FILL_A
21             LDAA #\0
22 0002         ENDMAC
23 0002         LABEL5: %FILL_A 23
    / 0002 8617   LDAA #23
24 0004         END

Errors:  None      #####
Bytes:   20        # DEMOXREF #
CRC:     0BE6      #####

```

Symbol and Cross Reference Table

=====

Symbol	Value	Type	Defline	Refline
--------	-------	------	---------	---------

Segment Definitions

CODE_1	0002	S00R	3	
CODE_2	0004	S03R	18	
OPER_STACK	0002	S01S	13	
SHARE_DATA	000C	S02C	16	

External Symbols

VALUE6	0000	X00	11	
--------	------	-----	----	--

Public Symbols

LABEL2	0000	R01E	14	15
VALUE2	001A	AE	5	6
VALUE4	001C	AE#	8	9

Local Symbols

LABEL1	0000	R00	12	
LABEL3	0000	R02	17	
LABEL4	0000	R03	19	
LABEL5	0002	R03	23	
VALUE1	0019	A	4	
VALUE3	001B	A#	7	
VALUE5	001D	V	10	

Macro Definitions

%FILL_A			20	23
---------	--	--	----	----

Also see 2.9 Assembler Listings.

6. Assembler Macros

This chapter describes the macro processing capabilities of the Archimedes A6801 Cross Assembler. The topics in this chapter include:

- An overview of the macro processor
- Basic macro usage, including macro definition directives, macro expansion and the use of substitution parameters; some simple example macros are included
- Advanced macro features, including the macro stack, the GENLAB facility, and automatic label generation
- Macro operator reference (for advanced macro use)

NOTE

All users who will be using macros should review 6.1 Macro Basics. Section 6.2 Advanced Macro Features contains information that will be of interest only to readers wishing to use the ADVANCED capabilities of the macro preprocessor, such as automatic label generation, the macro expression stack and advanced macro operators.

6.1 Macro Basics

This section serves as an introduction to the macro features of the Assembler, and explains how to define and use simple macros.

6.1.1 Macro Features

The A6801 Assembler includes a macro processing capability that can be used to create simple macros with up to 10 "substitution" parameters. However, it also contains many advanced features that can be used to implement high-level control structures made up of complex macro statements.

The A6801 macro facility includes the following features:

- Macro expansions can be nested, limited only by memory size (macro definitions cannot be nested, however)
- Macro definitions remain in effect throughout the entire source file in which they are declared, and can be used by any module within the source file
- A macro may accept up to 10 "substitution" parameters
- Unique local labels, and other incrementing values, can be generated automatically through the use of the "GENLAB" facility (advanced feature)
- A "macro stack" and advanced macro operators are available for implementing sophisticated macro functions

At the most basic level, the A6801 macro facility is very much like other macro processors. However, it is NOT directly compatible with any other assembler.

6.1.2 Macro Operation

In the simplest form, a macro is a user-defined symbol that represents a block of one or more Assembler source lines. Once you have defined a macro (using the MACRO and ENDMAC directives) you use it in your program like you would an Assembler directive or 68HC11 instruction. However, macros are handled differently by the Assembler than are directives or instructions.

When the Assembler finds a macro, it looks up its definition (which you have created) and expands the macro back into the source lines that it represents. The result is that the lines that the macro represents are, in effect, inserted into the source file at the location where the macro appeared.

Macros are similar in function to subroutines with one important difference: A macro causes code and/or data to be generated "in-line" in your program every time it is used (this is why this type of macro is often referred to as an "in-line" macro).

If a macro is defined as representing a block of code that ends up being 20 bytes long, that block will be inserted into the program every time the macro is used. If the macro is used 5 times, a total of 100 bytes of macro-generated code will be inserted into your program. (On the other hand, subroutines are "called" when needed, but the actual code for a subroutine only appears once in a program).

It is important to understand that the Assembler macro facility does not know (or need to know) anything about the 68HC11 target processor. This is because what it really does, for the most part, is to perform text substitution: It replaces the macro symbol with the block of Assembler source lines that it represents (and it also fills in the macro's parameter list, which we have not yet discussed).

There are also many advanced, high-level-language-like functions that macros can perform. However, these all result in some form of text substitution or text insertion as their final effect on the source code.

Lines generated through macro expansion are passed on to be assembled just as if all the lines came from the original source file.

The operation of the macro preprocessor will be further illustrated as we define some example macros and see how they are used. In the next section we will see how to define and use a simple macro.

6.1.3 A Simple Macro

The A6801 Assembler directive **MACRO** is used to define the name of a macro and to indicate the beginning of a macro definition. Please note that the definition for a macro must appear in the source file **BEFORE** the macro is used.

The syntax for the **MACRO** directive is shown below.

Syntax: [**<label>**] **MACRO** **%<name>**

A **MACRO** statement consists of an optional **<label>**, the **MACRO** directive itself, and the macro **<name>**, which must be immediately preceded with the percent character (%). The macro name is selected according to the standard rules for user-defined symbols (see 4.3 User-Defined Symbols).

The ENDMAC directive is used to mark the end of a macro definition, and is always the last line of the definition:

Syntax: [<label>] ENDMAC

We will now create a very simple macro to illustrate how these directives are used in a program:

```
MACRO    %STRING

FCB      'This is a macro-generated string'
FCB      $0D,$0A,0          carriage-return, linefeed, zero

ENDMAC
```

The MACRO statement assigns the name %STRING to this macro. The "%" character must appear as the first character in the name.

The FCB statements that follow form the body of the macro. They simply create some constant data in the form of an ASCII message terminated with a carriage-return, linefeed, zero sequence.

The body of a macro can consist of any valid A6801 source line: an assembler directive, an 68HC11 instruction, a comment, or a blank line. You may not, however, nest macro definitions by putting another MACRO statement in the body of a macro.

The ENDMAC statement marks the end of the macro definition.

In order to call this macro in your program, you simply type its name (including the % character):

```
.      JMP      SOME_ROUTINE
.
.      %STRING
.
```

When the Assembler encounters the %STRING symbol, it replaces it with the source lines that it represents (i.e., the body of the %STRING macro definition), so what gets assembled is:

```
.      JMP      SOME_ROUTINE
.
.      FCC      'This is a macro-generated string'
.      FCB      $0D,$0A,0          carriage-return, linefeed, zero
.
```

The end result is as though these lines had been part of the original source file.

Again, please note that a macro must be defined in a source file BEFORE it is called. Macro definitions can be placed at the beginning of the first module (after a NAME or MODULE statement) and will remain in effect throughout the entire source file, for all modules in that file.

The next section describes how macros can be enhanced with the use of macro parameters.

6.1.4 Macro Parameters

When a macro is called, up to 10 optional parameters may be specified along with the macro name itself on the source line. The syntax for a macro call that includes parameters is shown below:

Syntax: [<label>] MACRO %<name> [<p0>, <p1>, ..., <p9>]

The parameters represented by "p0" through "p9" can be referred to in the body of the macro definition as "\0" through "\9" (these are called the macro substitution operators).

When a macro is called, the Assembler takes the parameters supplied in the macro call (p0...p9) and inserts them into the expanded macro body at the positions indicated by the substitution operators (\0... \9). As each macro call can specify a different set of parameters, the expanded code can be different each time.

To illustrate this process, consider our earlier example to which we have added two macro substitution parameters. We have also called the %STRING macro twice, each time with a different set of parameters:

```
MACRO    %STRING    the macro body definition

\0: FCC    '\1'
FCB       $0D,$0A,0

ENDMAC

%STRING Msg1, This is a message #1       macro call #1
%STRING Msg2, This is a message #2       macro call #2
```

In this example, the macro substitution operators "\0" and "\1" will be replaced with the parameters specified in the macro call when the macro is expanded. The listing below shows what the two macro calls look like when they have been expanded:

```

9 0000                                %STRING Msg1, This is a message #1
/ 0000 54686973 Msg1:                FCC      'This is a message #1'
   0004 20697320
   0008 61206D65
   000C 73736167
   0010 65202331
/ 0014 0D0A00                        FCB      $0D,$0A,0
10 0017                                %STRING Msg2, This is a message #2
/ 0017 54686973 Msg2:                FCC      'This is a message #2'
   001B 20697320
   001F 61206D65
   0023 73736167
   0027 65202332
/ 002B 0D0A00                        FCB      $0D,$0A,0

```

Please note that in the first macro call "\0" has been replaced by "Msg1" and "\1" has been replaced by "This is message #1". A similar substitution has occurred for the second macro call.

6.1.5 Macro Guidelines

Please note the following in regards to using macros with parameters:

- A macro call can have up to 10 parameters, represented by "\0" through "\9" in the macro body
- The macro parameters may contain any characters, but a comma is assumed to be the delimiter between parameters (as it is between "Msg1" and "This is message #1" in the first macro call above)
- The end of the macro call parameter list is indicated by a comment delimiter (*) or the end of the source line (carriage-return)
- A null (missing) parameter in a macro call may be specified with two commas in a row (,,) as in:

```
%MYMAC      PARM1,,PARM3
```

- References in the body of a macro ("\n") to a parameter that was not supplied in the macro call will yield a null string

- When a macro parameter must contain a literal comma a special syntax must be used (because the comma is interpreted as a delimiter): The macro parameter must be surrounded by angle brackets (<>). For example, a call such as:

```
%D010    P0,<S,50>,,6
```

...would result in these parameter values in the body of the macro when it is expanded:

```
\0      =      P0
\1      =      S,50
\2      =      null string
\3      =      6
\4-\9   =      null string
```

- Macro parameter substitution only operates on literal string values. Therefore, numeric values may not be passed symbolically in a macro call. In the example below:

```
My_Number      EQU      12
```

```
%DEMOMAC My_Number
```

... the literal string "My_Number" (and not the defined value of 12) is passed to the macro.

6.2 Advanced Macro Features

This section discusses the advanced macro features of the A6801 Assembler, including the macro expression stack and the macro operators.

Also see the macro example in 6.2.5 Unique Labels with a macro.

6.2.1 The Macro Expression Stack

A global, static array of 32 bit integers is available for use by advanced macros, and is referred to as the macro expression stack. That this array is "global" means that it is available to all macros and that it is "static" means that its contents are not reset by the assembler from module to module within a single assembler source code file.

A stack pointer provides stack operation on this array (this is in no way related to the 68HC11 stack or stack pointer). The array may also be accessed by macros simply as an array of integers set aside for

their own use. Many of the special macro operators (described in 6.2.4 Macro Operator Reference) are used to manipulate this array.

6.2.2 Special Macro Symbols

To simplify the descriptions of the macro operators, we will use the following terms:

MSTACK

This is the macro expression stack (which consists of an array of 101 integers)

SP

This is the stack pointer for the macro expression stack, MSTACK. It is initially set to zero at the beginning of an assembly.

TOS

"Top Of Stack"* this is the MSTACK entry to which SP currently points.

NOS

"Next On Stack"* this is the MSTACK entry just "under" TOS. (TOS is MSTACK[SP] and NOS is MSTACK[SP-1].)

GENLAB

This is a global integer (separate from MSTACK) that is used for generating unique local symbols. It is initialized to zero at the beginning of each source code module.

6.2.3 Summary of Macro Operators

The following table summarizes the macro operators that are available in the A6801 Assembler:

Operator	Description	Return Value
\0 - \9	Macro parameters	parameter text
\?	Push # of parameters	new TOS
\\	Insert a literal backslash	"\" character
\Lnnn	Push a literal number	empty string
\@	Push and increment GENLAB	new TOS
\.	Get TOS without popping	TOS
\<	Get current TOS, pop stack	previous TOS
\O	Pop stack	new TOS
\A	Swap TOS & NOS	new TOS
\U	Duplicate TOS (to NOS)	TOS
\=	Set SP to TOS	new SP
\\$	Push SP	new TOS
\%	Push parameter length	new TOS
\"	Parameter substring	substring
\S	Store TOS indirect	stored #
\R	Recall TOS indirect	recalled #
\N	Negate TOS	new TOS
\I	Increment TOS	new TOS
\D	Decrement TOS	new TOS
\+	Add NOS to TOS	new TOS
\-	Subtract NOS from TOS	new TOS
*	Multiply TOS by NOS	new TOS
\/	Divide TOS by NOS	new TOS
\T	Suppress expansion if false	empty string
\F	Suppress expansion if true	empty string

The Return Value listed in the above table for each operator refers to the text string that replaces the operator when the preprocessor expands the macro.

Please note that macro operators are processed and expanded even when they are inside Assembler comments. This allows you to perform macro operations (such as accessing the macro stack or GENLAB) without generating code for the Assembler.

6.2.4 Macro Operator Reference

This section provides a detailed reference to each of the Macro Operators available in the Archimedes A6801 Assembler.

\0 - \9

These are the 10 macro substitution parameters. When a macro is expanded, the first parameter given in the macro call is available as \0, the next is available as \1, and so on.

Parameters specified in a macro call must be separated by commas (.). If a macro parameter must contain literal commas, then the entire parameter may be delimited by angle brackets (<>).

When a parameter is referenced in a macro definition, but that parameter is not supplied in the macro call, an empty string will be substituted for the reference (no error will be generated).

Given the following expansion call:

```
%TEST    1,Fred,,TEN,<A,B,C>,333
```

The referenced parameters would be substituted as:

\0	<--	1
\1	<--	Fred
\2	<--	(empty string)
\3	<--	TEN
\4	<--	A,B,C
\5	<--	333
\6 - \9	<--	(empty string)

NOTE

Macro parameters are always taken as literals in A6801. If you enter a DEFINEd, SET or EQUated symbol, the parameter will be expanded to the text of the symbol and NOT its value.

\?

Pushes onto the macro stack the number of parameters on the source line of the macro call.

SP <-- SP+1
MSTACK[SP] <-- # of parameters

Returns: the number of parameters (as a text string)

Returns a literal backslash character. For example, if you need a literal string '\%33' inside a macro, you must write it as '\\%33' in the definition. The second backslash will be removed when the macro is expanded.

\Lnnn and \Lnnn.

Pushes the numeric value nnn onto the stack. The numeric value may be in the range from 0 through 9999.

Notice that there are two forms of this operator: one ending in a period and one without a period. The operator ending with a period (\Lnnn.) allows it to be placed immediately before a numeric constant.

SP <-- SP+1
MSTACK[SP] <-- nnn

Returns: an empty string

\@

Pushes the value of GENLAB and then increments GENLAB.

To see how this might be used to automatically generate unique symbols see 6.2.5 Unique Labels with a Macro.

```
SP <-- SP+1
MSTACK[SP] <-- GENLAB
GENLAB <-- GENLAB+1
```

Returns: original value of GENLAB (as a text string).

\.

This operator expands into the number which is currently the top of the macro stack. It does not affect the content of the macro stack or the stack pointer.

Returns: TOS

\<

This operator expands into the number which is currently the top of the macro stack and then decrements the stack pointer.

```
SP <-- SP-1
```

Returns: original TOS

\O

This operator decrements the stack pointer and then expands into the number which is now the new top of the macro stack.

```
SP <-- SP-1
```

Returns: new TOS

\A

This operator exchanges TOS and NOS.

```
Tempval <-- MSTACK[SP]
MSTACK[SP] <-- MSTACK[SP-1]
MSTACK[SP-1] <-- Tempval
```

Returns: new TOS

\U

This operator makes a copy of the number in TOS and pushes it onto the stack.

```
SP <-- SP+1
MSTACK[SP] <-- MSTACK[SP-1]
```

Returns: new TOS

\=

This operator sets the stack pointer equal to whatever value is currently in TOS.

```
SP <-- MSTACK[SP]
```

Returns: the new TOS

\\$

This operator pushes the contents of the stack pointer onto the stack.

```
MSTACK[SP+1] <-- SP
SP <-- SP+1
```

Returns: the new TOS

\%

This operator takes TOS as the number of one of the macro expansion parameters (0 - 9). It looks at the text of that parameter and determines the length of its text. It then sets TOS to the string length of that parameter and returns the new TOS.

\"

This operator returns a substring from one of the macro parameters supplied by the macro call. Before using this operator, you must explicitly set up the macro stack (i.e., the MSTACK array) as follows:

MSTACK[SP] = parameter # (0-9) to take substring text from
MSTACK[SP-1] = beginning char position in parameter text
MSTACK[SP-2] = ending char position in parameter text

A parameter string is assumed to begin with character position 1.

If the specified beginning position (MSTACK[SP-1]) is larger than the ending position (MSTACK[SP-2]), the returned string will be reversed; if both pointers are 0, a null string will be returned.

SP is decremented by 3 to remove the substring specification from the stack.

Returns: the text of the requested substring

\S and \R

These two operators allow indirect storage and recall in the MSTACK array above the area currently in use as the macro stack:

\S

```
Tempval <-- MSTACK[SP]
SP <-- SP-1
MSTACK[MSTACK[SP] + SP] <-- Tempval
SP <-- SP - 1
```

Returns: Tempval (as a text string)

\R**MSTACK[SP] <-- MSTACK[MSTACK[SP] + SP]****Returns:** new TOS (as a text string)

(Note that \S and \R are not only indexed by TOS but are offset by SP.)

\N, \I, \D, \+, \-, * and \/

These seven operators perform integer arithmetic on the TOS (and NOS for binary operators).

The first three are unary operators, and do not affect SP:

\N: TOS <-- -TOS**\D: TOS <-- TOS-1****\I: TOS <-- TOS+1**

The last four are binary operators: they take both TOS and NOS as operands, they decrement SP and they leave their results in the new TOS:

\+: new TOS <-- old TOS + old NOS**\-: new TOS <-- old TOS - old NOS*****: new TOS <-- old TOS * old NOS****\/: new TOS <-- old TOS / old NOS****Returns:** all 7 return the new TOS (as a text string)

\T and \F

These two operators may cause the macro processor to cease expansion of macro operators on the current macro definition line depending on the value of the conditional assembly flag:

\T causes expansion to cease for the remainder of the line if the conditional assembly flag is FALSE.

\F causes expansion to cease for the remainder of the line if the conditional assembly flag is TRUE.

Returns: both \T and \F return an empty string

6.2.5 Unique Labels with a Macro

One of the most useful macro functions is using the GENLAB integer to automatically generate unique local labels for each expansion of a macro (see 6.2.2 Special Macro Symbols for an explanation of GENLAB).

The following listing (showing the macro expansions) illustrates how unique labels can be generated:

```

1 0000          MACRO %DECNSKIP
2              DECA
3              BNE TEMP@a
4              LDAA \0
5              TEMP\<: LDAB \1
6 0000          ENDMAC
7
8 0000          %DECNSKIP #10,#20
/ 0000 4A      DECA
/ 0001 2602    BNE TEMP0000
/ 0003 860A    LDAA #10
/ 0005 C614    TEMP0000: LDAB #20
9 0007          %DECNSKIP #11,#22
/ 0007 4A      DECA
/ 0008 2602    BNE TEMP0001
/ 000A 860B    LDAA #11
/ 000C C616    TEMP0001: LDAB #22
10 000E        END

```

The key to the above example is the "\@" operator, which returns the current value of GENLAB, and increments GENLAB, every time it is used (see the explanation of the "\@" operator). Since this operator is combined with "TEMP" on line #3, the unique symbols

"TEMP0000" and "TEMP0001" are created on the first and second macro calls, respectively.

Therefore, symbols created using the "\@" operator will have a unique number each time they are expanded.

In Line #5 of the macro definition we use the "<" operator to get a copy of the same number by the "\@" operator from the top of the macro stack in order to form the destination label on line #5 for the JNZ instruction on Line #3. Because this copy was retrieved from the macro stack, the value of GENLAB is not affected by "<" as it was by "\@".

6.3 Macro Examples

Shown below, are two more elaborated macro examples. The first example shows how to create a repeat function through the use of a recursive macro.

Example 1

```

1 0000          macro %repeat
2              lstexp -
3              if \0 > 0
4              lstexp -
5              rep_count set \0
6              endif
7              rep_count set rep_count-1
8              if rep_count>=0
9              lstexp +
10             lstexp +
11             \1
12             lstexp -
13             %repeat -1,\1
14             endif
15 0000          endmac
16
17 0000          %repeat 3,NOP
18 / 0000 00      NOP
19 / 0001 00      NOP
20 / 0002 00      NOP
21 0003          end

```

The second example shows how to create a variable macro through the use of a recursive help macro. This procedure is often required because the macro expander does not have direct access to the symbol table.

Example 2

```

1 * Macro example on how to get a variable macro through
2 * the use of a recursive help macro. This scheme is often
3 * required since the macro expander does not have direct
4 * access to the symbol table.
5
6 0000      MACRO %SET_MSTACK
7           LSTEXP -
8           IF \0>=0      INIT CALL?
9           *\T\L1
10          ?COUNT SET -(\0)
11          ENDIF
12          ?COUNT SET ?COUNT+1
13          IF ?COUNT<0
14          *\T\I
15          %SET_MSTACK -1
16          ENDIF
17          LSTEXP +
18 0000      ENDMAC
19
20 0000      MACRO %SUBSTR
21           LSTEXP -
22           %SET_MSTACK \1
23           %SET_MSTACK \0
24           *\L2
25           LSTEXP +
26           *   \"
27 0000      ENDMAC
28
29 0004      START SET 4
30 0007      ENDSTR SET 7
31 0000      %SUBSTR 2,4,ABCDEFGH
32 /         *   BCD
33 0000      %SUBSTR START,ENDSTR,ABCDEFGH
34 /         *   DEFG
35 0000      END

```

The Time Saver



TM

ARCHIMEDES
SOFTWARE

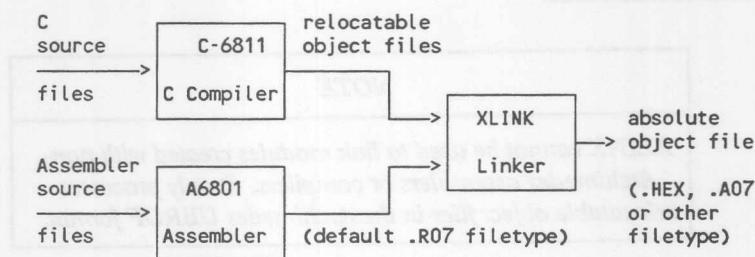
Linker

Linker

1. An XLINK Overview

The Archimedes XLINK cross-linker is a powerful, flexible software tool for use in the development of embedded-controller applications. XLINK reads one or more relocatable object files produced by the Archimedes A6801 assembler or C-6811 compiler and produces an absolute, 68HC11-executable program as output. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, mixed C-plus-Assembler programs.

The following diagram illustrates the linking process:



The object files (default .R07 filetype) produced by the A6801 assembler and C-6811 compiler use an Archimedes-proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format (see Appendix J, Glossary: UBROF). An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C. (The Linker processes .R07 files that come from Assembler and from the Compiler in the same way, and does not even need to know where they came from.)

XLINK performs three distinct functions when you "link" a program: it selectively loads modules containing executable code or data from the input file(s); it locates each segment of code or data at a user-specified address; and lastly, it links the various modules together by resolving all global (i.e., non-local, program-wide) symbols that could not be resolved by the Assembler or Compiler.

XLINK also supports user-created libraries. When XLINK reads a library file (which can contain multiple C or Assembler modules) it

will only load those modules which are actually needed by the program you are linking. This avoids having to load all the modules in a library file when you only need one routine. The XLIB Librarian is used to manage library files -- see the Librarian chapter of this manual for more information on libraries.

The final output produced by XLINK is an absolute, 68HC11-executable object file that can be burned into an EPROM, downloaded to a hardware emulator, or run directly on your host PC using a simulator/debugger. Through its support for over 30 different symbolic and non-symbolic object-file formats, programs produced by XLINK can be used with a wide variety of 68HC11 hardware emulators, software simulators, target boards and EPROM programmers. The most commonly-used output formats are Motorola (non-symbolic, for downloading to PROM programmers) and UBROF (symbolic, for use with most hardware emulators).

Lastly, XLINK can produce a detailed symbol cross-reference and segment/module "map" file that serves as a software debugging and documentation aide.

NOTE

XLINK cannot be used to link modules created with non-Archimedes assemblers or compilers. It only processes relocatable object files in the Archimedes UBROF format.

2. XLINK Characteristics and Features

XLINK is a fast, flexible development tool that boasts a number of important features and characteristics:

- Designed for modular construction of both C and Assembler programs
- Unlimited number of input files
- Fast, memory-based linking for most programs or optional disk-based operation for programs with large amounts of code/data or symbols
- Linker commands can be entered on the XLINK command line, read from an extended command file, or a combination of the two, making XLINK easy and flexible to use

- Searches user-defined library files and loads only those modules needed by the application
- Has the ability to typecheck "global" symbols at link-time, which assures that they are used consistently across all the modules (C or Assembler) in a program
- Symbols can be up 255 characters long, all positions significant, upper/lower case
- Global symbols can be renamed at link time so that a call to an as-yet-unwritten function can be "redirected" to a "stub" function (also useful for program configuration)
- Global symbols can be defined at link time
- Flexible segment commands allow full control of the locations of relocatable code and data in memory
- Supports bank-switched code with C and Assembler programs
- Rich set of commands and options for control of cross-reference listings
- Supports a large set (over 30) of industry-standard absolute object file formats, including Intel Hex (non-symbolic), Intel AOMF (symbolic), Motorola S-records (non-symbolic) and HP (symbolic)
- Supports C-language source-level debugging with certain 68HC11 hardware emulators (see Appendix H).
- Errors and warnings are expressed as complete messages
- DOS "environment variables" can be used to modify the defaults for various XLINK commands and options

3. Using the XLINK Linker

This Section serves as an introduction to using the XLINK Linker. It is not meant to be exhaustive in its coverage of every topic -- the discussion of some options is left to later sections in this chapter. The topics we will examine include:

- Installation
- File naming conventions used by XLINK
- Basic XLINK syntax
- Command line operation, with examples of linking small Assembler programs
- Command file operation, with example command (.XCL) files
- How segments, symbols, modules and libraries are processed by XLINK
- XLINK error handling
- The XLINK listing format

3.1 XLINK Installation

Before proceeding with this Section, you must install the XLINK software on your computer system. This involves creating directories on your hard disk and copying the necessary files from the distribution diskettes onto the system. Appendix-A contains installation instructions and lists the hardware requirements for your system.

IMPORTANT
<i>If you have not already done so, please install the XLINK software at this time on your computer as directed in Appendix-A: Installation.</i>

XLINK also supports a number of DOS "environment variables" that are used to specify default setup and command line parameters, which simplify linker operation. These variables are described in the XLINK Environment Variables section of this chapter.

3.2 File Naming Conventions

The XLINK Linker uses the following default naming conventions for the various files that it processes:

Filetype Description

- .R07 Relocatable files created by the A6801 Assembler or C-6811 Compiler that are read by XLINK
- .XCL XLINK Linker command files (default)
- .HEX Absolute, linked code in Intel-Standard "Hex" format, output by XLINK
- .A07 Absolute, linked code in symbolic "DEBUG" format (see Appendix J, Glossary: DEBUG) output by XLINK
- .LST Default filetype for cross-reference listings created by XLINK
- .MAP Preferred filetype for cross-reference listings created by XLINK
- .TMP Temporary file created if the "-t" option is used

The various output-file formats supported by XLINK use other filetypes in addition to those listed above. Please refer to the XLINK Output Formats section for a complete list.

Please note that the filetype ".MAP" is preferred for list files created by XLINK. This is because the default filetype of ".LST" might conflict with a ".LST" file created by the Assembler or Compiler, which would have the same name. As ".MAP" is not the default filetype, it must be specified explicitly using the "-l" command, described in the Essential XLINK Options section.

3.3 Basic XLINK Operation

This section describes basic XLINK command line operation and illustrates the use of the more common XLINK commands.

3.3.1 Command line Syntax

The general syntax for XLINK operation is shown below:

```
XLINK [-options...] input file(s) [-options]
```

...where

-options

Linker options and commands, which can be specified in any order, before or after the input file(s); these are listed and described in the next few sections and in the Linker Command Reference section.

input file(s)

A list of one or more relocatable object files to be linked into the program, in the order of appearance. The program you are linking can contain as many input files as desired. Each file can contain multiple modules, and each module can be loaded independently of the others. The section More About How XLINK Works later in this chapter discusses this process in more detail.

A drive and full pathname may be specified as part of the input file argument (e.g., "C:\PROJECT\LIB\BARFOO").

If XLINK is able to successfully complete a linkage, it finishes with a message as follows:

```
Archimedes Universal Linker V4.43/DXT  
(c) Copyright Archimedes 1992
```

```
Errors: none  
Warnings: none
```

If a linkage results in errors or warnings, XLINK will display a message for each error and show the totals number of errors and warnings in the summary fields above. For more information, see the "XLINK Errors and Warnings" section and Appendix-D: Linker Error and Warning Messages.

In general, there are two ways in which XLINK commands can be issued. In the Command line Mode, all input files and options are specified on a single XLINK command line. This method is best suited for linking small assembly-language programs with only a few modules and segments. Also, you may prefer this method as it gives you direct, command line control over all the linkage options.

In the Command File Mode of operation, input files and options are placed in an XLINK command file (default filetype of .XCL), with the filename specified on the XLINK command line with the "-f" option:

```
XLINK -f command file[.typ]
```

Large multi-module, multi-segment assembler programs, as well as all C programs, should always be linked using the command file (.XCL file) method, as entering all the necessary parameters on a single XLINK command line is impractical.

IMPORTANT!

The command file mode of operation is highly recommended when a large number of XLINK options must be specified, as when linking C-language programs.

Whether you use the Command line or Command File mode of operation, the same set of XLINK commands are used. The two modes of XLINK operation may also be mixed (i.e., arguments entered on the command line will be processed along with those read from an .XCL file).

NOTE for C-6811 Users

Please refer to the Archimedes C-6811 Compiler section for specific information on linking C programs.

NOTE for VAX, Sun & HP-3000 Users

XLINK users running on VAX/VMS, VAX/Unix, Sun or HP-3000 systems should see the next section, Notes for DEC, Sun and HP Users.

Before we look at some examples, please note the following guidelines for entering XLINK commands:

- If no input file(s) or options are specified on the command line, a "help" list of all the linker options will be displayed on your screen (along with the XLINK version number), as shown below:

Archimedes Universal Linker V4.43/DXT
(c) Copyright Archimedes 1992

```
Usage:      xlink {<options>} <sourcefiles> {<options>}
Environment: XLINK_ENVPAR
Options (specified order is of no importance):
-o file      Put object on: <file> <.extension>
-G          Disable global typecheck
-W          Disable warnings
-Z          Disable overlay check
-l file      Generate a list on: <file> <.lst>
-x{sme}      Generate cross-reference list
-pnn        Page listing with 'nn' lines/page (10-150)
-Dsymbols    Define global entries: ex. <symp a><,><symp
c><=><value>
-f file      Extend command line with <file> <.xcl>
-S          Silent operation of linker on terminal
-R          Disable range check
-ccpu       define cpu type
-Fformat     Define format type
-r{char}     Link debug system
-n          No local symbols
-Zsegmentlist Define segments: ex. <seg a> <,> <seg b> = <value>
-bbanklist   Define banked segments: ex. <seg a> <,> <seg b> =
<start>,<length>,<bankinc>
-A filelist  Forced load: ex. <filename> <,> <filename>
-B          Forced dump
-C filelist  Conditional load: ex. <filename> <,> <filename>
-E filelist  Empty load: ex. <filename> <,> <filename>
-d          Disable code generation
-Y{character} Extended format operation
-a{iw(.).l.l{.}} Overlay control
-m          Enable file bound processing
```

- t Enable temporary file
 - ereplacelist Define replace names for externals: ex. <extern to>=
<extern from>,<extern from2>
 - K Disable loading of library files
- The options and input file(s) may be specified in any order on the command line. The object files will be linked in the order they are specified on the command line (see the "More About How XLINK Works" section for more information), but the order of the options on the command line is of no significance.
 - An XLINK command line can be up to 127 characters in length (the MS-DOS limit). However, as mentioned, the "-f" option can be used tell XLINK to read additional arguments from a ".XCL" command file.
 - Upper/lowercase is of importance in two cases:
1. All command line options are case sensitive. For example, -f and -F are different options.
 2. All symbol and segment names are case-sensitive. For example the command "-Dmy_symbol=0" and "-Dmy_SYMBOL=0" define different symbols (the -D command is used to assign values to external symbols at link time).

IMPORTANT!

XLINK commands and user-defined symbols are CASE-SENSITIVE! DEC, Sun and HP-3000 Users -- please see the next section.

Input, output and listing filenames and arguments other than user-defined symbols may be entered in either upper or lower case.

- Spaces or tabs are the only valid delimiters that can be used to separate parameters. Commas (,) CANNOT be used as delimiters between parameters because they are used to separate items WITHIN an argument (as in the command "-ZSEG1,SEG2,SEG3"). The equals sign (=) is also used as a delimiter between items within some commands.

- The XLINK options that require a filename as part of the argument **MUST** have a space or tab between the option and the filename, as in the case of the "-l" (list-file) option:

```
-l myprog.lst      <-- space after -l
-o program.A07     <-- space after -o
```

Options that do not use a filename as an argument **MUST NOT** have a space or tab following the option, as in the case of the "-c" (CPU type) option:

```
-c68HC11           <-- no space after -c
-FMOTOROLA         <-- no space after -F
```

- When a number is used in an argument (e.g. an address or a value) it may be entered in decimal or hexadecimal format. A number preceded by a dot (.) is regarded as decimal, else it is interpreted as hexadecimal:

```
-Dsymbol=FF        <-- default format, hexadecimal
-Dsymbol=.255      <-- decimal format
```

3.3.2 Notes for DEC, Sun and HP Users

NOTE

This section only applies if you are using the VAX/VMS, VAX/Ultrex, Sun or HP-3000-hosted version of XLINK.

VMS on DEC VAX or Micro-VAX: Operation of XLINK under VMS is identical to MS-DOS except that all command lines are translated to lowercase by default by VMS. To enter an argument that must be in uppercase, you must enclose it in double quotation marks (" "). For example, if the following command was entered on an MS-DOS host system, the "-F" option must be entered in uppercase to differentiate it from the "-f" option:

```
xlink -c68HC11 program -FDEBUG
```

...so under VMS this command must be entered as follows:

```
xlink -c68HC11 program "-FDEBUG"
```

The double quotes disables the VMS case translation of command line input.

Unix on DEC VAX (Ultrix), Sun, or HP-3000: Operation of XLINK under Unix or Ultrix is identical to MS-DOS except that filenames that are case sensitive. This is significant when specifying input, output, listing and XLINK command filenames. For example:

FILE.XCL is NOT the same file as File.XCL

3.3.3 Essential XLINK Options

Out of 26 XLINK options, the following 8 are the most essential and commonly-used. A number of examples using each are presented in the next section:

NOTE

The complete set of XLINK commands is described in full detail in the XLINK Command Reference section.

-c68HC11

Defines the CPU type as 68HC11 or 68HC11-compatible. This option must always be specified when linking 68HC11 programs OR the "XLINK_CPU" environment variable must be set to "68HC11" (see the "Setting the Environment" section).

Typical usage:

-c68HC11

-Zsegment1[,segment2,segment3,...] [start address]

Locates a list of segments in memory, in the specified order, starting at address start address (addresses default to hex). XLINK locates the segments as follows:

segment1: begins at start address

segment2: begins at end of segment1 (start address + length of segment1)

segment3: begins at end of segment2 (start address + length of Segment1 + length of segment2)

... and so on, for all remaining segments

In other words, XLINK "stacks" the segments in memory, one after another, beginning with segment1 at location start address. You can specify as many separate -Z commands in a linkage as you need to define the location of all the various segments you are using in your program.

Typical usage:

-ZASM_CODE,CONST_DATA=0
-ZINTERNAL_DATA=8
-ZEXTERNAL_DATA=4000
-ZEEPROM_DATA=F000

If the -Z command is not specified, it is assumed that all code and data is "absolute", where the location of the code and data is specified in the Assembler source file with an ORG directive.

We have described only the most basic form of the -Z command -- there are many other ways in which it can be used.

-o file.[typ]

Puts the absolute output code in file.typ. If a -o command is not specified, the default output name of AOUT.A07 is used.

Typical usage:

```
-o CODE4U.A07
```

-z

Disables segment overlay checking. This option is needed for the 68HC11 because of its multiple address spaces, which allow multiple segments to reside at the same address, but in physically different memories (i.e., code, internal data, external data). If you attempt to link a program that has "overlapping" segments and do not specify "-z", XLINK will issue a warning message but should otherwise link the program correctly.

-l file.[typ]

This option is used to specify the name of the listing file (file.typ), and is normally used in conjunction with the "-x" option (below) which controls the contents of the listing. A default filetype is ".LST" is used if an extension is not specified. However, a filetype of ".MAP" is recommended so as to avoid confusion with assembler and compiler ".LST" files. "PRN" can be used to direct output to the printer. If a "-x" option is entered without "-l", the listing will display on the console.

Typical usage:

```
-x -l TOASTER.MAP
```

-x

Controls the contents of the listing file, which is directed to the file or device name specified in the "-l" command (above). When it is used by itself, "-x" tells the linker to include the following in the list file specified in the "-l" command:

- a header section with program information
- a module load map with symbol cross-reference
- a segment load map in dump order

Typical usage:

```
-l TOASTER.MAP -x
```

-Fformat

Sets the output format type. If this option is not specified, XLINK defaults to MOTOROLA for the output format. Another common format is DEBUG, which is a symbolic format used with most hardware emulators.

Typical usage:

```
-FMOTOROLA
```

-f command file

Extends the command line with command file.XCL. This allows you to place arguments in an ASCII ".XCL" file where they will be processed by XLINK (in addition to any commands specified on the command line). This is the preferred mode of XLINK operation, especially when linking C-6811 programs. See the "XLINK Command Files" section for example XCL files.

Typical usage:

```
-f BOB.XCL
```

NOTE for C-6811 Users

*"XCL" files should always be used to link C programs.
Please refer to the Archimedes C-6811 Compiler chapter
for specific information.*

3.3.4 Setting the Environment

Some of the XLINK options described above have settings that will seldom or never change during the course of your work. To make XLINK command line operation easier, we have provided a number of "environment variables" that can be used to supply default values for XLINK options (see Appendix J, Glossary: Environment Variables).

This section describes how to initialize three of these environment variables. The first environment variable you might want to set up is named "XLINK_CPU", and is used to make "68HC11" the default CPU type. This variable is initialized with the DOS "SET" command:

```
SET XLINK_CPU=68HC11
```

Now, whenever XLINK runs, it will use a default CPU type of 68HC11, so you do not need to enter "-c68HC11" on the command line or put it in a command file.

The next variable, "XLINK_FORMAT" sets the default format for the output object file to "DEBUG" :

```
SET XLINK_FORMAT=DEBUG
```

Most 68HC11 hardware emulators support the DEBUG format. If your emulator or development tool uses a different format, it should be entered here instead (see the "XLINK Output Formats" section).

Please note that the default CPU or output format setting will be overridden if a "-c" or "-F" argument, respectively, is specified on the command line.

To make the environment settings automatic whenever you boot your computer you can place the SET commands in your AUTOEXEC.BAT file (or your login script if you are running on a network).

NOTE

For a complete description of all the XLINK environment variables, please refer to the XLINK Environment Variables section.

3.3.5 Command line Examples

The following examples are used to illustrate the basic command line operation of XLINK. For our purposes here we will assume that we are linking simple assembly-language programs created with the A6801 assembler. C-language and larger assembly-language programs should be linked using the "command file" (.XCL) mode of operation, which is described in the XLINK Command Files section.

In all of the examples that follow, we assume that you have initialized the XLINK_CPU and XLINK_FORMAT environment variables as described in the previous section. (Please note that setting up these variables is equivalent to placing the "-c68HC11" and "-FMOTOROLA" arguments on the XLINK command line manually.)

NOTE

If you have not set up the DOS environment variables as explained in the previous section, or if you are running on a non-MS-DOS host (VAX, Sun or HP-3000), you must add -c68HC11 and -FMOTOROLA options to the example command lines.

REMEMBER

Upper/lower case IS significant for XLINK command line options.

XLINK DIPSTICK

This file links the object file DIPSTICK.R07 and creates an output file with the default filename of AOUT.A07. The output file will be in the MOTOROLA S-Record format as defined by the XLINK_FORMAT environment variable described above.

Please note that we have not given the linker any information about segments (i.e., no -Z option). This is only valid for "segment-less" programs -- those which contain only absolute (non-relocatable) code and data, whose address is determined at assembly time using ASEG and ORG directives.

If the file DIPSTICK.R07 contained relocatable segments (due to an assembler RSEG directives), the linker would issue a "Warning [8]: Segment segment undefined in segment or bank definition" in order to warn you that you have not located (defined) the segments. (However, it would go ahead and link the first segment at address 0 with any others following it).

Normally, your programs will contain one or more segments which need to be located, as in the following examples:

```
XLINK ZEBRA -ZCODE_A=C000 -ZDATA=2000 -o ZEBRA.A07
```

where, ZEBRA.R07 contains two segments, CODE A and DATA, both of which will be linked at address 0 where both ROM and external RAM begin (this is allowable in the 68HC11 hardware architecture). The output file would be named ZEBRA.A07 (-o option) and will be in the DEBUG format.

```
XLINK MONGOOSE -ZCODE_SEG1, CODE_SEG2=C000 -ZMDATA=2008 -o M.A07
```

... links the file MONGOOSE.R07 creating a DEBUG output file named M.A07 (-o option). The segment CODE_SEG1 will be located at address C000 while CODE_SEG2 will immediately follow it in memory. The segment MDATA will be located at location 2008.

Programs are often made up of multiple input files:

```
XLINK FILE1 FILE2 FILE3 FILE4 -ZROM=0 -ZRAM=8000 -o CONTROL.A07
```

... links together FILE1, FILE2, FILE3 and FILE4 (.R07) and places the absolute output file in CONTROL.A07.

Please note that when linking multiple input files, the order of filenames on the command line determines the order in which the files are loaded. For example, assuming that all the executable code is in segment "ROM", the code in FILE1 would start at address 0; the code in FILE2 would start at the next available address after FILE1; and so on, for FILE3 and FILE4.

The "-x" and "-l" options can be used to generate a listing file with a symbol cross-reference:

```
XLINK FRANK -ZCODE=0 -ZDATA=8 -l FRANK.MAP -x -z
```

... links FRANK.R07 and produces a listing file named FRANK.MAP (-l option). The "-x" option tells XLINK to include a module map with a cross-referenced symbol table and a segment load map. The Linker listing file can be used to determine the final location in memory of modules, segments and symbols as an aid for debugging and documentation. See the "XLINK Listings" section for an example of a Linker list file.

This last example illustrates the use of the "-f" option and the Command File mode of operation:

```
XLINK -f BIGPROJ -l BIGPROJ.MAP -x
```

... tells XLINK to read its arguments (input file names and options) from an ASCII file named "BIGPROJ.XCL". The command file mode of operation is functionally identical to command line mode except that many more input files and options can be specified, is less prone to error and easier to manage. The next section describes the use of Linker Command files in detail.

Please note that we have also specified a "-x" and a "-l" option on the command line. These arguments will be "added" to those being processed from the .XCL file.

3.4 XLINK Command Files

This section describes the use of the command file (.XCL) mode of XLINK operation.

3.4.1 Overview

As you recall, the "-f" command line option can be used to extend the XLINK command file by having it read arguments from a file. This is a very useful feature as it can often be impractical to enter all the required linker commands on one MS-DOS command line, even from a batch file. However, in all other ways, XLINK processes arguments from command files EXACTLY the same way as it processes command line arguments.

IMPORTANT!

We highly recommend that you always use XLINK command files when linking C-6811 programs as well as larger assembly-language programs. The C-6811 Compiler kit includes a number of "skeletal" Linker command files that should be used as the basis for your own .XCL files. See the Compiler chapter for details.

The general format for the command file mode of XLINK operation is as follows:

```
XLINK -f command file.[typ]
```

... where command file is the name of an ASCII text file that contains XLINK arguments (the names of the input file(s) and the Linker options). There must be a space or tab between the "-f" option and the filename. If the optional filetype is not specified, a type of ".XCL" is assumed (we often refer to these command files as "XCL files").

3.4.2 XCL File Guidelines

An XCL file can be created with any ASCII text or program editor. Arguments are written into the file exactly as you would type them on an XLINK command line. However, please note the following additional guidelines when writing your own XCL files:

- The end of each text line in the XCL file is treated as a delimiter between arguments (in addition to the spaces or tabs used in XLINK command line operation). You can place multiple arguments on one line, or put a single command on each line for readability. The .XCL file can be extended to as many lines as needed (to any size) to perform the link.
- A special "remark" delimiter (-!) can be used to surround comments so you can document complex XCL files:

```
-! Generate Motorola S-record Format -!  
-FMOTOROLA
```

Comments can be placed freely in the command file: on the same line following an XLINK argument (as above); on a line by themselves; or spanning multiple lines. The "-!" comment delimiter must be preceded by a space or tab.

WARNING!

The comment delimiter "-!" MUST be preceded by a space or a tab or must appear at the beginning of a line to be recognized as a delimiter. If you experience mysterious linker errors check the comment delimiters to be sure they are placed according to these rules.

- The use of an XCL file does not preclude the use of additional options typed on the XLINK command line, as in:

```
XLINK -f STDLINK FILE1 FILE2 -o PROGRAM.A07
```

This command causes XLINK to process the arguments in STDLINK.XCL (input files or options), load FILE1 and FILE2, then send the output to file PROGRAM.A07. In this way you could put all of your standard linker options and

library file loads in STDLINK.XCL and specify the other input and output files on the XLINK command line.

Command line arguments can be placed before or after the "-f" argument.

- An XLINK command line CAN contain multiple "-f" arguments. This allows you to process options and input files from more than one XCL file:

```
XLINK -f STDLINK -f PROJ1
```

XLINK will process input files and options first from STDLINK.XCL, then from PROJ1.XCL. This is a feature which can be used to create standard "sets" of XLINK commands or input files that you can select from as the job requires.

- Blank lines may be used in XCL files as desired to improve readability.

3.4.3 Example XCL Files

IMPORTANT for C-6811 Users!

The examples in this section are typical linker control files used to illustrate the general principles involved in writing your own XCL files. However, you should also review the Compiler documentation for version-specific details on XCL file creation and other requirements for linking C-6811 programs. The C-6811 Compiler Kit also includes a number of "skeletal" files that can be used as the basis for creating your own XCL files (e.g., LNK68HC11.XCL).

Example:

```
-! This is a simple XCL file to link ADEMO2.S07. -!  
  
ADEMO2      -! this is the input file to link -!  
  
-c68HC11 -z    -! sets the CPU type to 68HC11 and disables  
               overlay checking -!  
  
-! The next 2 lines locate our segments in memory -!  
  
-ZCODE,CONST_DATA=C000  -! code and constants go in ROM at C000 -!  
-ZDATA=2000             -! data at 2000H in 68HC11 internal RAM -!  
  
-o ADEMO2.A07          -! set the output file name -!  
-FMOTOROLA             -! set output format to Motorola -!  
-l ADEMO2.MAP           -! listing file name -!  
-x                     -! include xref segment/module map -!
```

Please note that it is perfectly allowable to place two or more arguments on one line (-c68HC11 -z). However, for readability, it is often desirable to write one command per line.

To invoke XLINK with this file you would enter:

```
XLINK -f ADEMO2
```

XLINK processes the commands exactly as if they had been entered entirely on the command line. It reads the entire ADEMO2.XCL file then processes the arguments, so the sequence of options is of no significance. Input files (if more than one) are loaded in the order which they appear (see the "More About How XLINK Works" section).

We will now look at a more advanced XCL file which controls the linking of the EXAMPLE.C program which is included with the ArchimedesC-6811 Compiler Kit. (Please note that your version of this file may be different than the one shown here). The embedded comments describe the function of each line:

```
-!                      -LNK6811.XCL-

XLINK 4.xx command file to be used with the 68HC11 C-compiler V3.x
using the -ms or -ml options (non banked memory models).
Usage: xlink your_file(s) -f lnk6811

First define CPU  -!

-c68hc11

-! Allocate segments which should be loaded -!

-! First allocate the read only segments.
C000 was here supposed to be start of PROM -!

-Z(CODE)RCODE, CODE, CDATA, ZVECT, CONST, CSTR, CCSTR=C000

-! Then the writeable segments which must be mapped to a RAM area
2000 was here supposed to be start of RAM.
Note: Stack size is set to 128 (80H) bytes with 'CSTACK+80'
!

-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK+80

-! The interrupt vectors are assumed to start at FFD6, see also
CSTARTUP.S07 and INT6811.H.

-Z(DATA)INTVEC=FFD6

-! NOTE: In case of a RAM-only system, the two segment lists may be
connected to allocate a contiguous memory space. I.e. :
-Z...CCSTR, DATA...=start_of_RAM -!

-! The segment SHORTAD is for direct addressing, let XLINK check
that the variables really are within the zero page -!

-Z(DATA)SHORTAD=00-FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write

-! See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read

-! Now load the 'C' library -!
cl6811 -! or cl6811d -!
example
asmfunc
```

```
-! Code will now reside in example.a07 in MOTOROLA 'S' format -!  
-o example.a07  
-! Create a map file with a full cross reference list -!  
-x  
-l example.map
```

This XCL file is typical for linking C programs. It loads relocatable input files: EXAMPLE.R07 (the main compiled C program), ASMFUNC.R07 (an assembly-language subroutine) and CL6811.R07 (the standard, large memory-model C-6811 library file). Only those modules that are actually needed from the CL6811 library file will actually be loaded by XLINK (see the "More About How XLINK Works" section).

The three "-Z" commands are used to locate the various code and data segments in the 68HC11 memory map. These segments (CSTACK, CSTART, DATA, etc.) are pre-defined by the C-6811 compiler and must ALWAYS be included when linking a C program. For more information on the use of these segments, refer to your Archimedes C-6811 Compiler User's Guide.

3.5 More About How XLINK Works

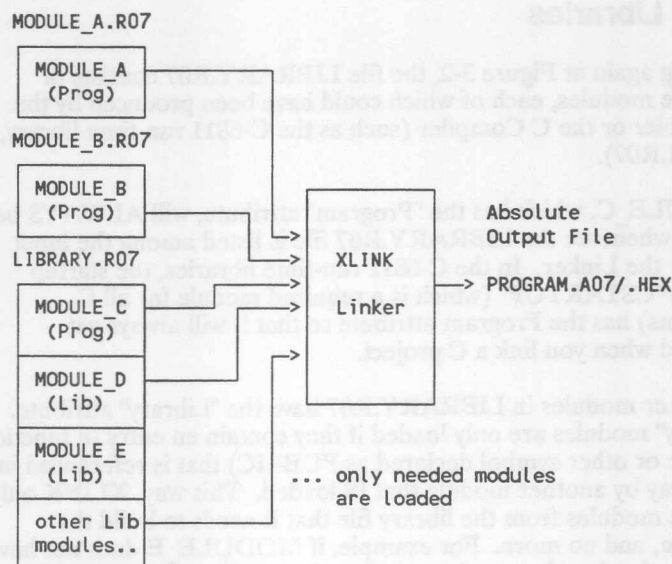
This section explains in more detail how XLINK processes its input files and modules, handles libraries, locates segments, resolves addresses and typechecks global symbols.

3.5.1 Input Files and Modules

The diagram below illustrates how XLINK processes input files and loads modules for a typical C-6811 program (although it could apply to larger multi-module assembler programs as well):

Relocatable Input

Files from Compiler (or Assembler)



Let us assume that the main program, for the sake of modularity and ease of maintenance, is made up of two source files, `MODULE_A.C` and `MODULE_B.C`, which have been compiled with C-6811 to produce relocatable ".R07" files. Each of these files consists of a single module ("MODULE_A" and "MODULE_B"). By default, the Compiler assigns the "Program" (Prog) attribute to both `MODULE_A` and `MODULE_B`. This means that they will ALWAYS be loaded and linked whenever the files they are contained in are processed by XLINK (i.e., the file names are given as arguments).

Please note that the code and data from a single C-6811 source file ends up as a single module in the .R07 file produced by the compiler. In other words, there is a one-to-one relationship between C source files and C modules. By default, the Compiler gives this module the

same name as the original (.C) source file (therefore, the source file `MODULE_A.C` produces a single module named "MODULE_A"). Libraries of multiple C modules can only be created using `XLIB` (see the Librarian chapter).

A6801 Assembler programs, on the other hand, can be constructed so that a single source (.S07) file contains multiple modules, each of which can be a "Program" module or a "Library" module.

3.5.2 Libraries

Looking again at Figure 3-2, the file `LIBRARY.R07` consists of multiple modules, each of which could have been produced by the Assembler or the C Compiler (such as the C-6811 run-time library, `CL6811.R07`).

`MODULE_C`, which has the "Program" attribute, will ALWAYS be loaded whenever the `LIBRARY.R07` file is listed among the input files for the Linker. In the C-6811 run-time libraries, the startup module "CSTARTUP" (which is a required module for all C programs) has the Program attribute so that it will always get included when you link a C project.

The other modules in `LIBRARY.R07` have the "Library" attribute. "Library" modules are only loaded if they contain an entry (a function, variable or other symbol declared as `PUBLIC`) that is referenced in some way by another module that IS loaded. This way, `XLINK` only gets the modules from the library file that it needs to build the program, and no more. For example, if `MODULE_E` does not have an entry that is referenced somewhere by another module, it will not be loaded.

The way this works is as follows: If the code in `MODULE_A` calls an external (non-local) routine or function, or makes a reference to an external symbol, `XLINK` will search the other input files for a module that contains that symbol as a "PUBLIC" entry (i.e., a module where the entry itself is located). If it finds the symbol declared as `PUBLIC` in `MODULE_C`, it will then load that module (if it has not already been loaded). Also see the "Symbols and Typechecking" section. It is important to understand that a "library" file is just like any other relocatable object (.R07) file. There is really no distinct type of file called a "library" (modules have a "Library" or "Program" attribute). What makes a file a "library" is what it contains and how it is used. Simply put, a "library" is an .R07 file that contains a group of related, often-used modules, most of which would have a Library attribute so that they can be loaded on a "demand-only" basis.

You can create your own libraries, or add to existing libraries, using C or Assembler modules. The C-6811 "-b" switch can be used to force a C module to have a Library attribute (instead of the default Program attribute). In Assembler programs, the MODULE directive is used to give a module a Library attribute (vs. the NAME directive, which gives it a Program attribute).

The XLIB Librarian is used to create and manage libraries. Among other tasks, it can be used to alter the attribute (Program/Library) of any module after it has been compiled or assembled. See the "Common Librarian Tasks" section in the Librarian chapter for more information.

Also see the following related Linker options in the Linker Command Reference section: "-d" (disable output of code); "-B" (force dump even with unresolved entries); "-A" (force load a Library module); "-C" (conditional-load a Program module as a Library module); and "-E" (load but do not dump a module).

3.5.3 Segment Location

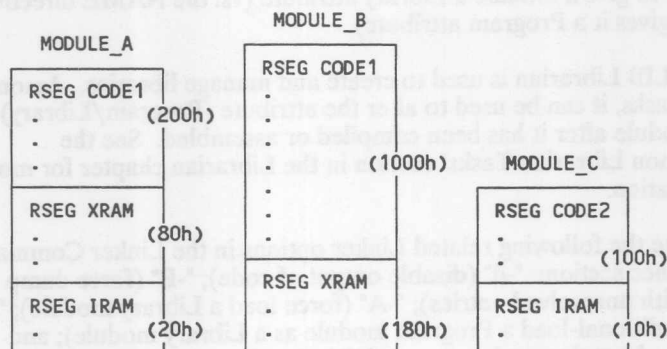
Once XLINK has identified the modules to be loaded for a program, one of its most important function is to assign load addresses to the various code and data segments that are being used by the program (see Appendix J, Glossary: Segment).

In assembly-language programs, the programmer is responsible for declaring and naming relocatable segments and determining how they are used (see the RSEG assembler directive in the Assembler chapter). In C-6811, the compiler creates and uses a set of pre-defined code and data segments. With C-6811, you have only limited control over segment naming and usage.

NOTE

For a list of the segments used by C-6811, see the Compiler chapter.

In order to understand how XLINK determines the final location of segments in 68HC11 memory, examine the diagram below which depicts the structure of an assembler program made up of three separate modules:



Each module contains "RSEG" directives that cause code or data to be allocated to one of four segments: CODE1 or CODE2 for executable code; XRAM for read/write variables stored in external RAM; and IRAM for data stored in 68HC11 internal RAM. The length of each segment within each module is shown in parenthesis (e.g., 100h bytes).

The following XLINK commands (placed in an XCL file) would be used to load and link these modules (assuming FILE_A contains MODULE_A, etc.):

```

.
.
FILE_A          -! load the input files/modules... -!
FILE_B
FILE_C
-ZCODE1, CODE2=C000 -! puts code segs where ROM starts, at C000H -!
-ZXRAM=2000        -! puts external data seg where RAM starts, at 2000
-!
-ZIRAM=0           -! start internal data seg at addr 0 -!
-x                -! generate a load map and listing file -!
-l FILE.MAP

```

The resulting segment/module load map for this hypothetical program would appear as shown below. This information is obtained from the linker listing file (for an example of an actual Linker listing, see the "XLINK Listings" section in this chapter).

Segment	Module	Load Address	Length
CODE1:	MODULE_A:	C000 - C1FF (ROM)	200h bytes
	MODULE_B:	C200 - D1FF "	1000h "
CODE2:	MODULE_C:	D200 - D2FF "	100h "
XRAM:	MODULE_A:	2000 - 207F (external RAM)	80h "
	MODULE_B:	2080 - 21FF "	180h "
IRAM:	MODULE_A:	0000 - 001F (internal RAM)	20h "
	MODULE_C:	0020 - 002F "	10h "

Note that the beginning address for each segment is as specified in the "-Z" command. The CODE2 segment is loaded at the next available address after the CODE1 segment, as they are listed together in the same "-Z" argument with a single starting address.

Within a given segment, code or data is loaded from the input modules in the same order as the input files to be loaded are specified in the XCL file or on the XLINK command line.

Also please note the following:

- If a start address is not specified in a "-Z" argument, XLINK will locate the segment at the next available address, or at 0 if any segment has yet to be defined:

```
-ZSEG1,SEG2=8000
-ZSEG3
```

If segment SEG2 ends at 3B02 then SEG3 will start at 3B03.

- Absolute code or data created in an A6801 assembler program following an ASEG directive is, in effect, "segment-less" -- the load address is determined by the Assembler based on the last ORG directive (or at 0 if an ORG has not been specified). XLINK has nothing to do with the placement of ASEG (absolute) code or data (see the ASEG directive in the Assembler chapter)

WARNING!

You must be careful not to locate absolute code or data at an address that will conflict with a relocatable segment or with other absolute code or data -- the Linker MAY NOT warn you if such a condition exists.

- A common mistake is attempting to use a "FCB" (Fill Code Byte) directive in a data segment (i.e., a segment that is located in RAM), which is not allowable -- initialized data constants can be placed in code memory (ROM) only. Normally, the only Assembler directive that should be used in a data segment is the "RMB" directive, which reserves space for data but does not cause initialization. Also see the "More About Segments and the 68HC11" section in the Assembler chapter for additional notes on using segments.

NOTE

The "-Z" command has many other forms of use -- see the detailed reference for this command in the XLINK "Command Detail" section for additional information. Also see the "-b" command for linking bank-switched segments.

3.5.4 Symbols and Typechecking

One of the most useful features of the XLINK Linker is that it performs global (program-wide) typechecking between modules at link time to be sure that functions are called and global data is accessed in a consistent fashion. This extra check helps detect a common source of program faults that would otherwise go undetected.

For example, consider the following situation where a C variable is declared in file TWO.C as a "long" but referenced in ONE.C as an external "int":

Source file ONE.C	Source file TWO.C
<code>extern int some_number;</code>	<code>long some_number;</code>
<code>void main()</code>	<code>void func()</code>
<code>{ ...</code>	<code>{ ...</code>

The C-6811 Compiler would not catch this type of error because it only checks individual source files. However, as long as the above example files were compiled with the "-g" Compiler option (which causes typechecking information to be included in the relocatable object files for all global symbols), XLINK would issue a warning when they are linked:

```
Warning[6]: Type conflict for external/entry some_number,
in module ONE against external/entry in module TWO
```

```
Errors: none
Warnings: 1
```

Typechecking is also performed on global C functions calls. The list of formal arguments for a function call will be checked for consistency along with the return type of the function.

For C programs, global typechecking is done automatically when the "-g" Compiler switch is used. For assembler programs, you must add your own typing information as part of the PUBLIC and EXTERN directives for the symbols you wish to typecheck. For information on how this is accomplished, see the PUBLIC Assembler directive in the Assembler chapter.

Also see the following related Linker options in the Linker Command Reference section: "-G" (disable global typechecking); "-e" (rename an external symbol at link time); and "-D" (define a global symbol at link time).

3.6 Linker Errors and Host Memory Management

This section discusses how the XLINK Linker handles error conditions in the link process and how to make best use of the host system's memory.

3.6.1 Errors and Warnings

NOTE
<i>See Appendix-D for a complete list of XLINK warning and error messages.</i>

The diagnostic messages produced by XLINK fall into five categories:

- Warning messages appear when XLINK detects that something might be wrong with the program (e.g., a segment definition is missing). However, the generated code may still be correct.
- Error messages are produced when XLINK determines that something in the program is definitely incorrect, but the error is not severe enough to cause the Linker to abort (e.g., two segments overlap). The output code is most likely faulty in some way.
- Fatal error messages are as error messages above, but cause XLINK to abort because it cannot complete the linking process because the fault is not recoverable (e.g., one of the specified input files is missing).
- Internal error messages are generated when, during the linking process, an internal test for program consistency fails to pass. The Linker will immediately abort after an internal error and display a short description of the problem.

Under normal operating conditions, you should never see an internal error message, but if you do, please report it to the Archimedes Software Technical Department and include as much information about the problem as possible (see Preface, How to Get Assistance).

- The memory overflow error condition occurs when XLINK runs out of host system memory. This error is described in the next section, Host Memory Management.

3.6.2 Host Memory Management

XLINK is a memory-based linker that may require as much as 570k of available RAM on a PC-DOS host to link a program. If it is run on a system with a small amount of available memory (< 400k), or if large files with many symbols are being linked, XLINK may run out of memory. This condition is indicated by a error special message:

```
*** LINKER OUT OF MEMORY ***
```

```
Dynamic memory used: nnnnnn bytes
```

... where nnnnnn is the number of bytes of RAM that XLINK was able to acquire for dynamic storage after it is loaded in memory (this is NOT the total amount of memory that XLINK finds in your system).

If a memory overflow occurs, apply one or more of the following cures, preferably in this order:

- Remove any TSR (memory-resident) programs (e.g. SideKick) or operating system drivers (e.g., disk cache, printer spooler) from your system.
- Some "MAKE" utilities can take up to 100k bytes of memory while they are running, so try running without MAKE.
- If you have less than 640k base RAM, install more memory to bring it up to 640k. In this case, use XLINK.OS2 instead of XLINK.EXE. To install the .OS2 version, rename the .EXE version to .SAV, then rename the .OS2 version to .EXE.
- If you have extended memory installed on your host system, use XLINK.EXE (not XLINK.OS2) which supports up to 16 MB of extended memory. Note that this version of Xlink does not support expanded memory(LIM, EMS, EEMS). This version of the linker is the default one.
- Use the "-m" XLINK option to enable "file-bound" processing, which causes XLINK to use a little less host system memory by using file pointers to all segments and modules instead of reading all the input files into RAM. XLINK will run more slowly if the -m option is used.

- Use the "-t" XLINK option to force symbols to be buffered in a temporary disk file (default name of XLINK.TMP) instead of in RAM. This can save even more RAM than the "-m" option but slows down XLINK even more.
- Reduce the number of global symbol definitions in a module to only those that are actually needed by that module. For example, it is a common programming practice to include a file with common global symbol definitions ("EXTERN" in assembler or "extern" in C) in every one of your source files. Unfortunately, while this is a convenient technique, it often results in a large number of external symbol declarations in every module that are never used, and every external symbol reference takes up memory in the Linker.

NOTE

Refer to the "XLINK Command Detail" section for a full description of the "-m" and "-t" options.

3.7 XLINK Listings

The XLINK Linker will produce a listing, as shown below, which serves as an aide for debugging and documentation. The listing shown below is from the ADEMO1.S07 assembly-language program included on the Assembler distribution diskette and listed in the Assembler chapter. This program is linked using the following XLINK command line:

```
XLINK ADEMO1 -c68HC11 -ZCODE=C000 -ZDATA=2000 -o ADEMO1.A07 -l  
ADEMO1.MAP -x
```

The "-l" option is used to specify the name of the file or MS-DOS device you want the listing to go to (ADEMO1.MAP in this case), while "-x" tells the linker to include a symbol cross-reference, module map and segment map.

```
#####
#
# Archimedes Universal Linker V4.38/DXT 27/Jan/92 09:09:07 [1] #
#
# Target CPU = 68hc11 [2] #
# List file = ademo1.map [3] #
# Output file 1 = ademo1.a07 [4] #
# Output format = motorola [5] #
# Command line = -c68hc11 -ZCODE=C000 -ZDATA=2000 -o ademo1.a07 #
# -l ademo1.map -x ademo1 [6] #
#
# (c) Copyright Archimedes 1991 #
#####
```

```
*****
*
* CROSS REFERENCE
*
*****
```

[7] Program entry at:C000 Relocatable, from module : ASM_DEMO_1

```
*****
*
* MODULE MAP
*
*****
```

```
FILE NAME : ademo1.r07 [8]
PROGRAM MODULE, NAME : ASM_DEMO_1 [9] [10]
```

SEGMENTS IN THE MODULE

```
=====
CODE [11]
Relative segment [12], address : C000 - C011 [13]
ENTRIES ADDRESS REF BY MODULE
BINHEX [14] C000 [15] Not referred to [16]
```

```
-----
DATA
Relative segment, address : 2000 - 2000
ENTRIES ADDRESS REF BY MODULE
BYTE_VAL 2000 Not referred to
```

```

*****
*
*          SEGMENTS IN DUMP ORDER
*
*****
[17]      [18]      [19]      [20] [21] [22] [23]
SEGMENT   START ADDRESS  END ADDRESS  TYPE  ORG  P/N  ALIGN
=====
CODE      C000    -    C011    rel  stc  pos    0
DATA      2000    -    2000    rel  stc  pos    0

*****
*
*          END OF CROSS REFERENCE
*
*****

```

Errors: none
Warnings: none

The first part of the listing is the PROGRAM HEADER, which contains:

- [1] Time and date of the linkage
- [2] The target CPU type (68HC11)
- [3] The output file or device name for the listing (ADEMO1.MAP)
- [4] The absolute output file name (ADEMO1.A03)
- [5] The output file format (MOTOROLA S-Record)
- [6] The full list of arguments (input files and options) specified for this link on the XLINK command line and in any XCL files that were specified with the "-f" option. The contents of any XCL files are shown in parenthesis ().

The second part is the CROSS REFERENCE, which consists of two sub-sections, the MODULE MAP and the SEGMENT MAP (SEGMENTS IN DUMP ORDER).

- [7] The program entry point (0000 in module ASM_DEMO_1) is a special entry that is used in some output formats for hardware emulator support; see the END Assembler directive in the Assembler chapter for an explanation of how this is specified and used.

The MODULE MAP consists of a sub-section for each module that was loaded as part of the program. Each sub-section includes:

- [8] The name of the input file that the module came from (ADEMO1.R07)
- [3] The module type (PROGRAM, vs. LIBRARY)
- [10] The module name (ASM_DEMO_1)
- [11] A list of the segments in each module, with the following information listed for each:
 - [12] The segment type: relative, stack, banked or common (see the "Segment Directives" section in the Assembler chapter); the segments are relative in our example because the RSEG directive was used
 - [13] The segment load address, specified as a range in hex
 - [14] The global symbols that have been declared within each segment (BINHEX, BYTE_VAL,...)
 - [15] The final, absolute location in memory (a hex address) for the global symbol.
- [16] The names of other module(s) in the link that refer to this symbol

The SEGMENTS IN DUMP ORDER SECTION section lists all of the segments that make up the program, shown in the order that they were linked. For each segment the map includes the following information:

- [17] User-defined name of the segment
- [18] Starting address of the segment, in hex
- [19] Ending address of the segment, in hex
- [20] The segment type: relative (rel), stack (stc), banked (bnk) or common (com); see the "Segment Directives" section in the Assembler chapter.
- [21] The segment's origin: whether the start address of the segment is static/absolute (stc) or floating (flt); the segment that is listed first in a "-Z" argument which includes a load address is "absolute"; the rest of the segments in the argument are "floating" because their address depends upon the size of the segment(s) that precede them.
- [22] Positive/Negative allocation (P/N): whether the segment is allocated upwards (pos) or downwards (neg) in memory; see the "-Z" command in the "XLINK Command Detail" section.
- [23] Segment alignment: how the segment is aligned in memory; see the "Segment Directives" section in the assembler chapter for more information on this option.

Also see the following related Linker options in Section 10: Linker Command Reference: "-p" (controls the number of lines/page); and "-x" (controls the contents of the listing).

Section Summary

In this Section we have looked at all of the essential aspects of XLINK Linker operation, including installation, file naming, command line and command file operation with examples, module and segment processing, error handling and Linker listings.

The next Section contains a detailed reference to all of the XLINK commands. You should review this Section at least once so that you are aware of all the Linker options that are available.

4. Linker Command Reference

This Section contains a detailed reference on the use of all the XLINK Linker commands and options. It assumes that you already have an overall understanding of XLINK operation, as covered in the previous section: Using XLINK. This Section includes:

- XLINK Command Syntax
- Summary listing of all XLINK command line options, organized by functional category
- XLINK Environment Variables
- Detailed description for each XLINK option, listed in alphabetical order:

-! Comment Delimiter	-K Default Libraries
-A Force-load Input	-l List File ControlFile
-b Banked Segments	-m File-bound Processing
-B Forced Dump Option	-n Local Symbol Option
-C Conditional-load	-o Output File ControlFile
-c CPU Selection	-p Page Listing Control
-D Define Symbols	-r Special Debug Format
-d Disable Code Gene	-R Range Check Option ration
-e Rename a Symbol	-S Silent Linker Option
-E Empty-load Input	-w Disable Warning Messages File
-f Extended Command	-x Cross Reference File
-F Output Format Sel	-z Disable Overlay Checking ect
-G Global Typechecking	-Z Segment Definition

IMPORTANT for C-6811 Users!

Please review your Compiler chapter for version-specific information regarding Linker usage that may not be covered in this Section. Also, the C-6811 Kit includes various "skeletal" Linker command (.XCL) files that you should use as the basis for your own .XCL files. The Compiler Chapter of this manual describes these files.

4.1 XLINK Syntax Reference

4.1.1 General Syntax

The general syntax for XLINK operation is:

```
XLINK [-options...] input file(s)... [-options...]
```

...where

-options

Linker options/commands described in this Section. These may be specified in any order, and may be freely mixed with the list of input file(s). Spaces or tabs separate options.

input file(s)

A list of one or more relocatable object files (default filetype of .R07) to be linked into the program. A program can contain as many input files as desired, and XLINK will load them in the order they are listed within a given segment. A full drive and pathname may optionally be specified as part of the input file argument.

Each input file can contain multiple modules. A module with a "Library" attribute will be loaded independently of the other modules in a file if it has an entry (public symbol) that is referenced in another loaded module.

4.1.2 Delimiters

A space or a tab serves as the delimiter between arguments.

The XLINK options that require a filename as part of the argument **MUST** have a space or tab between the option and the filename, as in the case of the "-l" (list-file) option:

```
-l myprog.lst
```

<-- space after -l

Options that do not use a filename as an argument **MUST NOT** have a space or tab following the option, as in the case of the "-c" (CPU type) option:

-c68HC11

<-- no space after -c

4.1.3 Defaults

The default filetype for relocatable object input files is .R07.

If an output file is not specified (see the "-o" option in the "XLINK Command Detail" section), a default output file named AOUT.A07 will be created in the default output format MOTOROLA S-Record.

If no input file(s) or options are specified on the command line, an abbreviated "help" list of all the linker options will be displayed on your screen, as well as the Linker version number.

4.1.4 Upper/Lower Case

Upper/lowercase is of importance in two cases:

1. All command line options are case sensitive. For example, -f and -F are different options.
2. All symbol and segment names are case-sensitive. For example the command "-Dmy_symbol=0" and "-Dmy_SYMBOL=0" define different symbols (the -D command is used to assign values to symbols at link time).

IMPORTANT!

XLINK commands and user-defined symbols are CASE-SENSITIVE! DEC, Sun and HP-3000 Users -- please see section 4.1.8 for important information.

Input, output and listing filenames and arguments other than user-defined symbols may be entered in either upper or lower case.

4.1.5 Command Files

An XLINK command line can be up to 127 characters long. However, XLINK is most often used in a command file mode of operation where input files and options are read from an ".XCL" file:

```
XLINK -f command file[.typ]
```

If a filetype is not specified for the command file, ".XCL" is assumed. Additional input files and options can also be specified on the same line, where they will be added to those in the ".XCL" file. Multiple "-f" commands may also be specified.

4.1.6 Numeric Formats

When a number is used in an argument (e.g. an address or a value) it may be entered in decimal or hexadecimal format. A number preceded by a dot (.) is regarded as decimal, else it is interpreted as hexadecimal:

```
-Dsymbol=FF          <-- default format, hexadecimal  
-Dsymbol=.255        <-- decimal format
```

4.1.7 Errors and Warnings

For a list of Linker warning and error messages, see Appendix-D:Linker Error/Warning Messages.

4.1.8 Notes for DEC, Sun and HP Users

NOTE

This section only applies if you are using the VAX/VMS, VAX/Ultrix, Sun or HP-3000-hosted version of XLINK.

VMS on DEC VAX or Micro-VAX: Operation of XLINK under VMS is identical to MS-DOS except that all command lines are translated to lowercase by default by VMS. To enter an argument that must be in uppercase, you must enclose it in double quotation

marks (" "). For example, in the following command, the "-F" option must be entered in uppercase to differentiate it from the "-f" option:

```
xlink -c68HC11 program -FMOTOROLA
```

...so under VMS this command must be entered as follows:

```
xlink -c68HC11 program "-FMOTOROLA"
```

The double quotes disables the VMS case translation of command line input.

Unix on DEC VAX (Ultrix), Sun, or HP-3000: Operation of XLINK under Unix or Ultrix is identical to MS-DOS except that filenames are case sensitive. This is significant when specifying input, output, listing and XLINK command filenames. For example:

```
FILE.XCL is NOT the same file as File.XCL
```

4.2 XLINK Command Summary

Input/Output File Options

-A file,...	Force-load input file(s) as Program module(s)
-B	Force XLINK to output code even with errors
-C file,...	Conditional-load input file(s) as Library module(s)
-d	Disable all output code generation
-E file,...	Empty-load input file(s) (load but don't output)
-Fformat	Select absolute output format
-o file	Set output file name to file
-Y{01}	Controls the end record of the Intel_standard format

Segment Control Options

-bbank_def	Define banked segments according to bank_def list
-Zseg_def	Define segments according to seg_def list
-z	Disable all overlay checking
-a{i,w,[.],{.}}	Segment overlay control

Symbol Control Options

- Dsym=val Define symbol sym with value val
- enew=old,old Rename old symbol names(s) to new
- G Disable the global typechecking feature
- n Disregard all local symbols when linking

Listing Control Options

- l file Set list file name to file (PRN for printer)
- pnnn Set page size to nnn lines/page (10-150)
- x{eims} Generate cross-reference (e=entry map, i=indexed list, m=module map, s=segment map)

Command File Options

- f file Specify extended command line (XCL) file
- #! comment -! Comment delimiter for XCL files

Miscellaneous XLINK Options

- ccpu Select processor type cpu
- K Disable loading of "default" libraries
- m Enable "file-bound" processing to save memory
- R Disable the CPU address range check
- S Enable "silent" (no messages) linker operation
- t Use a temporary file for symbols to save memory
- w Disable linker warning messages

Notes for MS-DOS Users

This section describes the use of the MS-DOS environment variables and error return codes supported by XLINK.

4.2.1 XLINK Environment Variables

XLINK uses a number of DOS environment variables (see Glossary) which can be defined in the PC host environment using the DOS "SET" command. These variable can be used to create defaults for various XLINK options so they do not have to be specified on the command line. (Also see the "Setting the Environment" section in this chapter and Appendix-A: Installation).

Except for the XLINK_ENVPAR and XLINK_TFILE environment variables, the default values can be overruled by the corresponding command line option.

For example, the "-FMPDS" command line argument will supercede the default format selected with the XLINK_FORMAT environment variable.

To make these settings automatic, you can place the SET commands in your system's AUTOEXEC.BAT file (or in your "login script" in you are running on a network).

XLINK_PAGE

Sets the number of lines per page (20 - 150). The default is a non-paged list. Also see the -p option in the "XLINK Command Detail" section.

Example

```
SET XLINK_PAGE=64
```

XLINK_COLUMNS

Sets the number of columns per line in the list file. The default is 80 columns.

Example

```
SET XLINK_COLUMNS=132
```

XLINK_CPU

Sets the target CPU type. Also see the -c option in the "XLINK Command Detail" section.

Example

```
SET XLINK_CPU=68HC11
```

XLINK_FORMAT

Sets the output format. Also see -F option in the "XLINK Command Detail" section and the XLINK Output formats section.

Example

```
SET XLINK_FORMAT=DEBUG
```

XLINK_MEMORY

If set to zero (0) the linker is file bound, else it is memory bound. Also see the -m option the "XLINK Command Detail" section.

Example

```
SET XLINK_MEMORY=0
```

XLINK_ENVPAR

Creates a default XLINK command line which contains normal XLINK arguments.

Example

```
SET XLINK_ENVPAR=-DCONF=7 -t
```

XLINK_TFILE

Sets the name and location of the "temporary" file which is used when the -t command is specified. See the -t option in the "XLINK Command Detail" section for more information.

Example

```
SET XLINK_TFILE=E:\XLINK.TMP
```

4.2.2 DOS Error Return Code

XLINK returns status information to DOS which can be tested in an MS-DOS batch file using the "IF ERRORLEVELn" statement. The supported error codes are listed below:

- 0 Linking successful
- 1 There were warnings generated during the link (unless the XLINK -w option is specified, in which case XLINK will return a 0 on warnings)
- 2 There was a non-fatal error
- 3 Fatal error detected (XLINK aborted)

These error codes can be used as follows in a batch file:

```
.
XLINK -f TESTLNK
IF ERRORLEVEL 3 GOTO ERRORS
IF ERRORLEVEL 1 GOTO WARNINGS
ECHO The Link was successful!
.
:WARNINGS
ECHO The Link ended with a warning or non-fatal error...
.
:ERRORS
ECHO The Link ended with a fatal error...
.
```

NOTE

Please refer to your MS-DOS documentation for more information on the use of environment variables, error return codes and batch files.

4.3 XLINK Command Detail

This section consists of a reference to each of the XLINK commands and options, which are listed in alphabetical order.

-!

Comment Delimiter

Syntax

-! comments... -!

Description

The "-!" option is used to bracket comments within an XLINK Extended Command line (XCL) file. Comments may be placed on the same line following an XLINK argument or can be used on a line by themselves. A comment may also span several lines as long it is enclosed by "-!".

NOTE

A space or tab character MUST precede the "-!" comment delimiter when it is not placed at the beginning of the line.

Example

```
-! TIMER.XCL: This XCL file is used to control  
the link process for the TIMER.C program. -!  
-c68HC11 -! set the CPU type for the 68HC11 -!  
.
```

See Also the "XLINK Command Files" section in this chapter.

-A**Force-load Input Files****Syntax**

-A input file, input file,...

Parameters

input file(s): The relocatable object file(s) to be force-loaded; the default filetype is .R07. A full drive and directory path can be specified along with the file name(s).

Description

This command temporarily forces all of the modules within the specified input file(s) to be loaded as if they were all "Program" modules, even if some of the modules have the "Library" attribute.

This option is particularly suited for testing library modules before they are installed in a library file since the -A option will override an existing library module with the same entires. In other words, XLINK will load the module from the input file specified in the -A argument instead of one with an entry with the same name in a Library module.

Example

```
-! these lines are in an .XCL file... -!  
-A PUTCHAR  
CLIB
```

These two lines cause XLINK to load the user-written PUTCHAR.R07 library module instead of the one residing in the CLIB library (assuming that the PUTCHAR file contains the same global entry as one of the modules in CLIB). Please note that this is only true if the "-A PUTCHAR" command appears before the "CLIB" input file in the .XCL file.

See Also

- *-C (Conditional-Load Input Files) option*
- *The XLIB Librarian chapter*
- *Glossary: Program and Library Attributes*

-a**Control Segment Overlay****Description**

The -a (disable overlay of local areas) linker option now provides control of overlaying of the local area of each function. The -a option operates in conjunction with a new overlay algorithm as described in the READ.ME file.

The -a option now has 5 modifiers as described below:

Syntax

`-a{i, w, (x,x,...x), [x,x,...x], {x,x,...x} }`

Where each "x" is the name of a global function or local function. A local function may specify a module name by using the form "module:name", i.e. the module name prepended to the function name, separated by a colon.

-a linker option modifiers:

`nothing` Disable all overlay of local areas.

`i` Disable overlay of indirectly called function trees with each other.

`w` Disable the multiple root warning (Warning [16]).

`(x,x,...,x)` Do not overlay the function trees of these functions with other function trees.

`[x,x,...,x]` Do not allocate the function trees of these functions if they are not called by another function. If you have unfinished functions, use this to tell the linker that these are not interrupt functions.)

`{x,x,...,x}` These functions are interrupt service routines.

NOTE

Once you have added a modifier to the -a option, you cannot disable it (e.g. with another -a line later in the link).

-b**Define Banked Segments****Syntax**

-b[adr type][(type)]seg list = start,length,increment

Parameters

adr type_{opt}: Specifies the type of load addresses that XLINK will use when it dumps the output code: "#" for linear physical addresses; "@" for 64180-type physical addresses; or nothing for logical addresses with bank number. Please note that only a limited number of output formats support banked addresses (see the "Linker Output Formats" section).

type(opt): Specifies the type for all segments specified in seg list: CODE, DATA, XDATA, IDATA, BIT or UNTYPED (the default if none is specified). See the -Z command for more information on how these types are used.

seg list: List of banked segments to be linked. Packing of the segments is dependent on the delimiter between segments in the bank list. If the delimiter after a segment is a colon (:) the next segment will be placed in a new bank; if it is a comma (,) the segment will be placed in the same bank as the previous segment.

start: The start address of the first segment in seg list. This is a 32-bit value: The high-order 16 bits represent the starting bank number while the low-order 16 bits represent the starting offset address within the bank.

length: The length of each bank, in bytes. This is a 16-bit value.

increment: The incremental factor between banks, i.e., the number that will be added to start to get to the next bank. This is a 32-bit value: the high-order 16 bits are the bank increment, and the low-order 16 bits are the offset increment.

(All numbers are hex by default. Use a period before the number to specify a decimal base, as in ".32".)

Description

The `-b` command is used to allocate banked segments for a program that is designed for bank-switched operation. It also enables the "banking" mode of linker operation.

Example

```
-b(CODE)BSEG1,BSEG2,BSEG3=8000,4000,10000
```

This line specifies that the three code segments should be linked into banks starting at 8000, each with a length of 4000, with an increment between banks of 10000.

See Also -Z (Define Segments) option

NOTE

The C-6811 compiler supports bank-switching for program code. Please see the Compiler chapter for more information on using this feature.

-B Force-Dump Output File**Syntax**

-B

Description

This option causes the Linker to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered. (XLINK always aborts on fatal errors, even with -B.)

The -B option could be useful in cases where missing entries will be "patched" in later in the absolute output image.

See Also the "Errors and Warnings" section in this chapter

-C Conditional-Load Input Files**Syntax**

-C input file, input file,...

Parameters

input file(s): The relocatable object file(s) to be conditionally-loaded; the default filetype is .R07. A full drive and directory path can be specified along with the file name(s).

Description

This command temporarily causes all of the modules within the specified input file(s) to be treated as if they were all "Library" modules, even if some of the modules have the "Program" attribute. This means that the module(s) in the input file(s) will be loaded only if they contain an entry that is referenced by another "loaded" module.

One application for the -C option is for testing the C-6811 "CSTARTUP" module before installing it in the library (see Example).

Example

```
-! these lines are in an .XCL file... -!  
CSTARTUP  
-C CL6811
```

This example assumes that the user-created CSTARTUP file contains the CSTARTUP module, which is going to load instead of the "Program" module of the same name residing in the file CLIB. In effect, the CSTARTUP in CLIB will be forced to "Library", allowing the user-created version to load. Please note that this is only true if the "CSTARTUP" input file appears before the "-C CLIB" command in the .XCL file.

See Also

- -A (*Force-Load Input Files*) option
- *The XLIB Librarian chapter*

-C

Select CPU Type

Syntax

```
-ccpu
```

Parameters

cpu: Processor type, from one of the following:

z8	z80	z8002	68HC11
8048	8085	8086	1802
6801	6805	6803	68k
t33	t7000	16032	6502
8036	7800	6301	68hc11

Description

This command sets the CPU type. You must specifying "68HC11" for 68HC11-family chips as well as for the various proliferation devices.

The environment variable XLINK_CPU can be SET to install a default for the -c option so it does not have to be specified on the command line.

Example

-c68HC11

See Also "XLINK Environment Variables" (this Section)

-D**Define Symbols****Syntax**

-Dsymbol = value

Parameters

symbol: Any external (EXTERN) symbol in the program that is not defined elsewhere.

value: The value to be assigned to symbol.

Description

The -D switch allows the definition of absolute symbols (entry) at link time. This is especially well suited for configuration purposes. Any number of symbols can be defined using the XCL file mode of XLINK operation. The symbol(s) defined in this manner will belong to a special Linker-generated module named ?ABS_ENTRY_MOD. XLINK will display an error message if you attempt to re-define an existing symbol.

Example

```
-Dversion=.41  
-Dmonitor_entry=F010
```

The first example sets version to 41 decimal (the period indicates decimal notation). The second example could be used to set the address of the entry point to a monitor routine in ROM (in hex).

See Also "Symbols and Typechecking" section

-d**Disable Code Generation****Syntax**

-d

Description

The -d option disables the generation of output code from XLINK. Otherwise, XLINK performs as usual. This option is useful for "trial" linking of programs (e.g., checking for syntax errors, missing symbol definitions, etc.). XLINK will run slightly faster for larger programs when this option is used.

-e**Rename Symbols****Syntax**

```
-enew name=old name,old name,...
```

Parameters

new name: A new name you wish to assign to an existing external symbol. Usually, this is a public symbol that represents an existing function or data address.

old name: One or more existing external symbol names that you wish to rename, separated by commas (,).

Description

Replaces external symbol(s) *old name(s)* with new *name*. This command is useful for configuring a program at link-time as it can redirect a function call from one function to another function. This can also be used for creating "stub" functions (i.e., when a system is not yet complete, undefined function calls can be directed to a "dummy" routine until the real function has been written).

Example

```
-esmall_printf=printf
```

This example will cause all calls to function "printf" to be redirected to "small_printf" (which must be a valid public function).

See Also the "Symbols and Typechecking" section

-E

Empty-load Input Files

Syntax

-E input file, input file,...

Parameters

input file(s): The relocatable object file(s) to be "empty-loaded"; the default filetype is .R07. A full drive and directory path can be specified along with the file name.

Description

This command causes the listed input file(s) to be "empty-loaded", i.e., processed normally in all regards by the Linker but output code will not be generated. One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by "empty

loading" all input files except the one(s) you want to appear in the output file.

Example

```
-E FILE2,FILE3  
FILE4  
-O PROJECT.HEX  
.
```

In this example, only the modules from FILE4 will appear in the PROJECT.HEX output file, but the program will otherwise link normally.

-f

Extended Command line (XCL)

Syntax

-f command file[.typ]

Parameters

command file: The name of an ASCII file that contains arguments to be processed by XLINK. If a filetype (.typ) is not specified, .XCL is assumed. A full drive and directory path can be specified along with the file name.

Description

The -f option is used to extend the XLINK command line by causing it to read arguments from a command file, just as if they were typed in on the command line. Arguments are entered into the XCL file with a text editor using the same syntax as they would be typed on the command line. However, in addition to spaces and tabs, the end-of-line (carriage-return) is also treated as a valid delimiter between arguments.

The "-" comment delimiter can be used to bracket comments placed in the file for documentation.

Example

See the "XLINK Command Files" section

IMPORTANT for C-6811 Users!

Please review the Compiler documentation for version-specific details on XCL file creation and other requirements for linking C-6811 programs.

The C-6811 Compiler Kit also includes a number of "skeletal" files that can be used as the basis for creating your own XCL files (e.g., LNK68HC11.XCL).

-F**Select Output Format****Syntax****-Fformat****Parameters**

format: One of the supported XLINK output formats from the following list:

intel-standard	intel-extended	motorola
millennium	ti7000	rca
symbolic	typed	ashling
pentica-a	pentica-am	pentica-ai
msd-i	msd-m	msd-t
msd	mpds	mpds-i
mpds-m	mpds-code	mpds-symb
hp-symb	hp-code	hp
tektronix	extended-tekhex	aomf8096
aomf8051	nec-symbolic	nec
ashling-6301	ashling-6801	ashling-64180
ashling-z80	ashling-8085	ashling-8080
debug	zax	zax-i
pentica-b	pentica-bm	pentica-bi
pentica-c	pentica-cm	pentica-ci
pentica-d	pentica-dm	pentica-di

Description

The -F command is used to select the format for the absolute output file created by XLINK. This allows you to use a wide variety of development tools to debug programs linked with XLINK. If a -F option is not specified, the default of MOTOROLA S-RECORD is used.

The environment variable XLINK_FORMAT can be SET to install an alternate default format on your system. See "Environment Variables" at the beginning of this Section.

For the 68HC11, by far the two most commonly-used formats are: MOTOROLA S-RECORD, a non-symbolic, ASCII format which is readable by almost any PROM programmer, hardware emulator, target board or simulator; and DEBUG, a binary, symbolic format used by many hardware emulators.

Some of the supported formats include symbolic debugging information, while others include just the code itself. In certain cases, two output files are created (one for code, one for symbols). The format selected often determines the default filetype used for the output file(s). See the "XLINK Output Formats" section for details on the output formats.

Example

- FMOTOROLA

NOTE

Please review the "Linker Output Formats" section for complete details on selecting the correct output format for use with your hardware emulator or other development tool.

See Also

- the "Linker Output Formats" section
- -o option (Set Output-File Name)
- -Y option (modify output format)

-G

Disable Global Typechecking

Syntax

-G

Description

By default, XLINK performs link-time typechecking between modules by comparing the external references to an entry

with the public entry itself (if the information exists in the object modules involved). A warning is printed if there are mismatches, but the Linker will continue otherwise and not abort.

The -G option disables this typechecking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

Please note that C-6811 programs must be compiled with the "-g" option in order for link-time typechecking to work. With assembly-language programs, the typing information is specified by the programmer as part of the PUBLIC and EXTERN directives (see the "Assembler Directive Reference" section in the Assembler chapter).

See Also the "Symbols and Typechecking" section

-K

Disable Default Library Load

The "default library" option is not utilized by the Archimedes C Compilers, so this XLINK option is not currently supported.

-l

Set List File Name

Syntax

-l list file[.typ]

Parameters

list file: Specifies the name of the file or device to which a listing is directed. If a filetype is not specified, ".LST" is used by default. However, a filetype of ".MAP" is recommended so as not to confuse Linker list files with Assembler or Compiler list files. A full drive and directory path can be specified along with the file name.

Description

The **-l** option is used to set the name of the file or device to receive the Linker listing. A device name of "PRN" or "LPTn" (where n is 1, 2 or 3) will cause the listing to be sent directly to the named printer.

The **-x** option is used to control the contents of the listing. If **-l** is used without **-x**, only the basic program "header" information will be included in the listing. If **-x** is specified, the listing will contain a cross-reference, a module map and a segment load map (see the **-x** command for other options).

Example

```
-l SYSTEM1.MAP -x
```

This example creates a listing file named SYSTEM1.MAP which includes a cross-reference/load map (**-x**).

See Also

- **-x** (*Generate Cross Reference*) option
- **-p** (*Set Page Length*) option
- The "XLINK Listings" section

-m

Enable File-Bound Processing

Syntax

```
-m
```

Description

The **-m** option causes XLINK to use less host system memory by using file pointers to all segments and modules instead of reading all input files into RAM. If XLINK runs out of host memory during a link, this option will often help. However, XLINK will run more slowly if the **-m** option is used.

The environment variable XLINK_MEMORY can be SET to "0" have XLINK default to file-bound processing mode without having to specify the -m option. See the "Environment Variables" section in this chapter.

See Also

- -t (Temporary File) option
- the "Host Memory Management" section

-n

Disregard Local Symbols

Syntax

-n

Description

This option causes XLINK to ignore all local (non-public) symbols it finds in the input modules. If -n is used, locals will not appear in the listing cross-reference and will not be passed on to the output file. Using this option speeds up the linking process and can also reduce the amount of host memory needed to complete a link.

Note

Local symbols are only included in files compiled with the "-g" or "-r" C-6811 options, assembled with the "S" command line option, or "LOCSYM+" Assembler directive.

IMPORTANT!
<i>The -n option must NOT be used if you are debugging with a hardware emulator that supports source-level debugging. These tools require that local symbols be present in the DEBUG output file.</i>

See Also the "Symbols and Typechecking" section

-o**Set Output File Name****Syntax**

-o output file[.typ]

Parameters

output file: Name of the file to contain the linked, absolute output code. A full drive and directory path can be specified along with the file name.

Description

The **-o** command is used to specify the name of the XLINK output file. By default, the Linker will use the name AOUT.A07 if a name is not supplied. If a name is supplied without a filetype, the default filetype for the selected output format (**-F** option) will be used. For the MOTOROLA S-RECORD and DEBUG formats, the default filetype is ".A07".

If a format is selected that generates two output files, the user specified filetype (.typ) will only affect the primary output file (first format).

Example

```
-o \PROGRAMS\FINAL.A07 -FMOTOROLA
```

This example creates an output file "FINAL.A07" in directory "\PROGRAMS\" using the MOTOROLA format (**-F**).

See Also

- **-F** (Set Output Format) command
- the "XLINK Output Formats

-p**Listing Page Length****Syntax**

-pnnn

Parameters

nnn: Number of lines per page, specified as a decimal number (10-150).

Description

Sets the page length for the XLINK listings in lines per page. The default is 80 lines/page. The environment variable XLINK_PAGE can be SET to install a default page length on your system. See the "Environment Variables" section.

Example

-p66

-r{char}**Enable the DEBUG Format****Purpose**

The -r option causes the linker to produce the Archimedes proprietary symbolic DEBUG/UBROF output format. This option should only be used with debuggers that accept this format. Note that this option overrides the -F linker option.

-R

-R

If an address is relocated out of the 68HC11 CPU's address range (code, external data or internal data address), an error message is generated. This usually indicates an error in an assembly-language module or in the XLINK segment definition list (-Z command).

-S

The **-S** option turns off the **XLINK** sign-on message and final statistics report so that nothing appears on your screen while it runs. However, it does not disable error and warning messages or the listing output.

See Also -w (Disable Warning Messages) option

-t**Enable Temporary File****Syntax**

-t

Description

The -t option forces XLINK to use a temporary file (with the default name XLINK.TMP in the current directory) to store a large part of the linker symbol tables. This can significantly reduce the amount of host system memory needed to link a program with a large number of symbols (> 1500). In some cases, it may be necessary to use this option to complete a link process.

Note that the -t option can significantly increase the time it takes to link a program. The -m (Enable File Bound Processing) option will also automatically be enabled when -t is used.

The environment variable XLINK_TFILE can be SET to an alternate file name (with drive and directory path) to use for the temporary file. See "Environment Variables" at the beginning of this Section.

See Also

- *-m (Enable File-Bound Processing) option*
- *the "Host Memory Management" section in this chapter*

-w**Disable Warning Messages****Syntax**

-w

Description

This option causes XLINK to suppress all warning messages, although they will still be counted and will be shown in the Linker final statistics.

See Also

- the "Errors and Warnings" section in this chapter
- Appendix D.2: XLINK Warning Messages

-x**Generate Cross-Reference****Syntax**

-x[ems]

Parameters

- e**: Generate a module entry list
- m**: Generate a module map
- s**: Generate a segment map

Description

The **-x** option is used to control the contents of the XLINK listing file. When **"-x"** it is specified without any of the optional parameters, a default cross-reference listing will be generated which includes (this is equivalent to **"-xms"**):

- a header section with basic program information
- a module load map with symbol cross-reference
- a segment load map in dump order

If the **-x** option is succeeded by one or more of the arguments **e**, **m**, or **s**, the listing will be modified as follows:

The **"e"** option causes an "entry" map to be included in the cross-reference listing, which consists of an abbreviated list of every entry (global symbol) in every module. This "entry map" is useful for quickly finding the address of a routine or data element.

The "m" option causes a full module map to be generated, which lists all segments, local symbols and entries (public symbols) for every module in the program (see the example in the "XLINK Listings" section in this chapter).

The "s" option causes a segment load map to be included, which consists of a list all the segments in "dump" order (see the example in the "XLINK Listings" section).

If a -l (Set List File Name) option is not specified in the link, the listing will be sent to your screen.

Example

```
-xems -l PROGRAM.MAP
```

This example generate a cross-reference listing with an entry map, module map and segment map to the file PROGRAM.MAP.

See Also

- -o (Set List-File Name)
- the "XLINK Listings" section

-Y{0,1, 2}

Modify Output Format

Pentica and MSD-SYMB

When using the Pentica -a, -b, -c, -d and MSD-SYMB formats (specified with the -F linker option) you can also use the -Y option in conjunction with the -F as follows:

- Y0 option enables all symbols to be module:symbolname
- Y1 option enables local and line symbols to be module:symbolname
- Y2 option to enable line symbols to be module:linenumber.

Pentica-a, -b, -c, -d now uses three-byte addresses for addresses that are more than two bytes.

AOMF51 for Hitex Emulator

To generate code for the Hitex emulator for the 8051, use the -Faomf8051 and -Y0 options.

Intel-standard

-Y{01} switch used to control the end record in the Intel-standard format. When **-Y** (default **-Y0**) is used, the Hex file is always ended with **:00000001FF**. However, when the **-Y1** is used, the Hex file is ended with the program entry record if it exists; otherwise it is ended with **:00000001FF**.

MPDS-CODE

The MPDS-CODE format can fill unused bytes with either 00s or FFs. By default, the linker fills these bytes with 00s. If filling unused bytes with FFs is required, the user must link using the **-Y0** switch.

-Z**Disable Overlay Checking****Syntax**

-Z

Description

By default, XLINK checks to be sure that the various segments that have been defined (**-Z** command) do not overlap in memory.

-Z**Define Segments****Syntax**

`-Z[(type)]segment,[segment,...][=#][range,range,...]`

Parameters

type: Optional. Specifies the type for all segments specified in segment list as one of: CODE, DATA, XDATA, IDATA, BIT or UNTYPED (the default if none is specified). See below.

segment(s): List of segments to be linked, with a comma used as a delimiter between each one. The segments are allocated in memory in the same order as they are listed. Note: Appending "+nnnn" to a segment name causes an increase in the amount of memory that XLINK will allocate for that segment by nnnn bytes. See Examples below.

"#" or "=" (allocation operator): Optional. Specifies how the segments are allocated. If "=" is used, the segment(s) are allocated so they begin at the start address in the specified range (upwards allocation). If "#" is used, segment(s) are allocated so they end at the end addresses in the specified range (downwards allocation). If an allocation operator (and range) is not specified, the segments will be allocated upwards from the last segment that was linked, or from address 0 if no segments have been linked. See Examples below.

range(s): Optional. If present, this defines the location in memory where the listed segment(s) will be placed. This is done with a list of addresses and/or address ranges separated by commas. An address range is defined as: "start address - end address". See Examples below.

(All numbers are hex by default. Use ".number" to specify a decimal base, as in ".32".)

Description

The **-Z** command is a powerful XLINK option that allows you to determine how and where segments will be allocated in the overall memory map.

A list of segment(s) can be specified in one **-Z** command, in which case the segments will be allocated in the specified order at the specified address range. Multiple **-Z** command lines can be used in a single linkage, as needed.

If XLINK finds a segment in an input file that is not defined either with **-Z** or **-b** (banked segment definition command), a warning will be displayed by the Linker. However, the segment will still be allocated as if it were listed in the last segment definition (i.e., at the next available address).

Additional related topics and optional forms for **-Z** are described below.

Segment Typing

The optional type parameter is used to assign a "type" to all of the segments in the list. It is important to understand that this "type" parameter does not affect how XLINK processes the segments -- it only generates information in the output file that is used by some hardware emulators (those that use one of the formats listed below) to identify which address space (from the list below) a public symbol belongs to.

The supported segment types are listed below:

CODE	contains code
DATA	contains generic data
UNTYPED	the default if type is not specified
XDATA	external data
IDATA	internal data
BIT	bit data

The use of the segment type field is limited by the output format being used (**-F** option). **CODE**, **DATA** and **UNTYPED** are only relevant when one of the following

output formats are being used. Only hardware emulators that fully support one of these output formats will recognize the segment typing information:

extended-tekhex	ashling-z80
aomf8051	ashling-64180
ashling-6301	hp-symb
ashling-6801	nec-symb
ashling-8080	nec
ashling-8085	

Relative, Common and Stack Segments

There are three different types of segments that can be processed by XLINK: Relative, common and stack (see the "Segment Directives" section in the Assembler chapter to see how these segments created).

Stack segments grow, by default, from high to low addresses. The load address is subtracted by the (aligned) segment size before the allocation is performed. Succeeding segments in the list will be loaded below the preceding segment. (However, if executable code is placed in a stack segment it will not be put in reversed order as data would). Relative and common segments are by default, allocated from low to high addresses (upwards allocation). However, you can change the allocation direction at link time with the -Z command using the "=" (upwards) or "#" (downwards) operator.

If stack segments are mixed with relative or common segments in a segment definition, the Linker will produce a warning message but will allocate the segments according to the default allocation set by the first segment in the segment list.

Common segments have a size equal to the largest declaration found for the particular segment. That is, if module "A" declares a common segment "COMSEG" with

size 4, while module "B" declares this segment with size 5, the latter value will be allocated for the segment.

Relative and stack segments have a size equal to the sum of the different (aligned) declarations.

Examples

NOTE

These examples illustrate the basic concepts involved in the -Z command. C-6811 users should refer to the Compiler manual and "skeletal" XCL files for additional -Z examples.

-ZSEGA,SEGB=0

... locates SEGA at address 0, followed by SEGB which starts after the end of SEGA.

-ZSEGA,SEGB#1000

... segment SEGA will be allocated downwards from 1000H, followed by SEGB below it.

-Z(IDATA)SEGA+100,SEGB=0

... will increase the space reserved for segment SEGA by 100 bytes (hex).

-ZSEGA,SEGB=100-200,400-700,1000

... SEGA will be placed between address 100 and 200, if it fits in that amount of space. If it does not, XLINK will try the range 400 - 700. If none of these ranges are large enough to hold SEGA, it will start at 1000.

SEGB will be placed, according to the same rules, after segment SEGA. If SEGA fits the 100 to 200 range then XLINK will try to put SEGB there as well (following SEGA). Otherwise, SEGB will go into the 400 - 700 range if it is not too large, or else it will start at 1000.

-ZSEGA,SEGB+10,SEGC+10=10-50,60

... increase both SEGB and SEGC by 10 (hex). If SEGA fits into the memory area 10 - 50, it will be located there. If not, it is located at 60. The linker then tries to allocate SEGB and SEGC, starting with SEGB. When linking is complete, the segments will be in this order: SEGA, SEGB and, at the highest address, SEGC.

-ZSEGA,SEGB,SEGC#10-50,30

... SEGA will be allocated, if it fits, between 10 and 50 (hex), ending at address 50 (since the "#" operator specifies downward memory allocation). If not, the segment will end at 30 (hex). Then SEGB and SEGC will be allocated, with SEGB just before SEGA and then SEGC, just before SEGB.

If these ranges are too short, it will cause either error [24] "Segment segment overlap segment segment", if any segment overlaps another, or error [26] "Segment segment is too long", if the ranges are too small. SEGA will be at the highest address, then SEGB, and then SEGC with the lowest starting address.

See Also

- the "Segment Location" section
- the "Segments" (defining in an assembler program) section
- the "More About Segments and the 68HC11" section

5. XLINK Output Formats

This Section is a reference for the various output formats supported by XLINK with the "-F" command. It also outlines which output format(s) are appropriate for use with the various 68HC11 hardware emulators, PROM programmers, and target boards.

The contents of this Section include:

- Summary list of XLINK output formats
- Summary of which output formats to use with the 68HC11 hardware emulators, PROM programmers and target boards
- A detailed description of output formats

5.1 Summary of XLINK Output Formats

The following two tables summarize the Linker output formats that can be specified in the "-F" XLINK command (see the -F command in the "XLINK Command Detail" section). The first table lists the formats that result in the creation of a single output file, while the second list the formats that result in two output files.

Format (specified in -F)	Format Type	Default Filetype	Address Type
AOMF8051	Binary	.A07	N
AOMF8036	Binary	.A07	N
ASHLING	Binary	none	N
ASHLING-6301	Binary	.A07	N
ASHLING-64180	Binary	.A07	NS
ASHLING-6801	Binary	.A07	N
ASHLING-8080	Binary	.A07	NS
ASHLING-8085	Binary	.A07	NS
ASHLING-Z80	Binary	.A07	NS
DEBUG	Binary	.DBG	NL
EXTENDED-TEKHEX	ASCII	.A07	NLPS
HP-CODE	Binary	.X	N
HP-SYMB	Binary	.L	N
INTEL-STANDARD	ASCII	.A07	N
INTEL-EXTENDED	ASCII	.A07	N
MILLENNIUM (Tektronix)	ASCII	.A07	N
MOTOROLA	ASCII	.A07	NLPS
MPDS-CODE	Binary	.TSK	N
MPDS-SYMB	Binary	.SYM	NLPS
MSD	ASCII	.SYM	N
NEC-SYMBOLIC	ASCII	.SYM	N
PENTICA-A	ASCII	.SYM	NLPS
PENTICA-B	ASCII	.SYM	NLPS
PENTICA-C	ASCII	.SYM	NLPS
PENTICA-D	ASCII	.SYM	NLPS
RCA	ASCII	.A07	N
SYMBOLIC	ASCII	.A07	NLPS
TEKTRONIX (Millennium)	ASCII	.HEX	N
TI7000 (TMS7000)	ASCII	.A07	N
TYPED	ASCII	.A07	NLPS
ZAX	ASCII	.A07	NLPS

In the above table, the Format is the name which is specified in the XLINK "-F" command. These Formats are detailed in the "Output Format Details" section in this chapter. The Format Type refers to whether ASCII or binary encoding is used for the format. The Default Filetype is shown for each format, which is used if a filetype is not specified as part of the XLINK "-o" command (see -o in the "XLINK Command Detail" section in this chapter). Finally, the

Address Type indicates whether the format supports banked addresses, and if it does, what type of banked address:

- N = Nonbanked address
- L = Banked logical address
- P = Banked physical address
- S = Banked 64180 physical address

The next table lists the output formats that cause XLINK to generate two different output files, one with code and one with symbolic information for use by a hardware emulator:

Format (specified in -F)	Code Format	Default Filetype	Symbolic Format	Filetype
HP	HP-CODE	.X	HP-SYMB	.L
MPDS	MPDS-CODE	.TSK	MPDS-SYMB	.SYM
MPDS-I	INTEL-STANDARD	.HEX	MPDS-SYMB	.SYM
MPDS-M	MOTOROLA	.S13	MPDS-SYMB	.SYM
MSD-I	INTEL-STANDARD	.HEX	MSD	.SYM
MSD-M	MOTOROLA	.HEX	MSD	.SYM
MSD-T	MILLENNIUM	.HEX	MSD	.SYM
NEC	INTEL-STANDARD	.HEX	NEC-SYMB	.SYM
PENTICA-AI	INTEL-STANDARD	.OBJ	PENTICA-A	.SYM
PENTICA-AM	MOTOROLA	.OBJ	PENTICA-A	.SYM
PENTICA-BI	INTEL-STANDARD	.OBJ	PENTICA-B	.SYM
PENTICA-BM	MOTOROLA	.OBJ	PENTICA-B	.SYM
PENTICA-CI	INTEL-STANDARD	.OBJ	PENTICA-C	.SYM
PENTICA-CM	MOTOROLA	.OBJ	PENTICA-C	.SYM
PENTICA-DI	INTEL-STANDARD	.OBJ	PENTICA-D	.SYM
PENTICA-DM	MOTOROLA	.OBJ	PENTICA-D	.SYM
ZAX-I	INTEL-STANDARD	.HEX	ZAX	.SYM
ZAX-M	MOTOROLA	.HEX	ZAX	.SYM

In the above table, Format is the name which is specified in the XLINK "-F" command. The Code Format is the actual output file format used for the first file, which contains code and constant data (these formats are detailed in the "Output Format Details" section). The Default Filetype for the first (code) file is shown for each format, which is used if a filetype is not specified as part of the XLINK "-o" command (see -o in the "XLINK Command Detail" section). The primary filename for the first file comes from the name specified in the "-o" command ("AOUT" is used as a default if a name is not specified).

The Symbolic Format is the format used for the second file, which contains symbolic information used by hardware emulators. The Filetype shown for this file is not a default, but is a fixed filetype that is always used. The primary filename for the second file comes from

the name specified in the "-o" command ("AOUT" is used as a default if a name is not specified).

Please see the "-Z" (segment location) command in the "XLINK Command Detail" section for more information on how the typing information is specified.

5.2 Emulator/Development Tool Support

Refer to the EMULATOR.DOC file on the distribution diskettes for a list of hardware emulators for the 68HC11.

PROM Programmers

Typically, the MOTOROLA S-RECORD format is supported with most PROM programmers for "burning" EPROM chips. Refer to the PROM programmer documentation for details on how to download the resulting Hex file from your host system to the programmer.

Some PROM programmers will accept only "binary image" files (i.e., a file that contains an exact load image of the linked program, padded with zeros where there are "holes" in the image). The MPDS-CODE output format creates a straight "binary dump" file of this type.

5.3 Output Format Details

This section contains a detailed reference to the output formats supported by XLINK.

AOMF8051

This is the Intel AOMF (Absolute Object Module Format) for the 8051. It is a binary format that includes information for symbolic debugging and segment typing.

This is the most commonly-used 8051 symbolic format, and is compatible with the Archimedes SimCASE Simulator/Debugger and most 8051 hardware emulators.

This is an Intel-proprietary object file format.

When this format is used with a Hitex Emulator, the -Y0 linker option must be used.

AOMF8096

This is the Intel AOMF (Absolute Object Module Format) for the 8096. It is a binary format that includes information for symbolic debugging and segment typing.

This is an Intel-proprietary object file format.

ASHLING

For explanations see the Ashling document "Ashling Microsystems Object Module Format ver 1.0.1".

DEBUG

This format is a binary format that include information for source level debugging and segment typing. It is mostly used by 68HC11 hardware emulators.

This is an Archimedes proprietary format.

Extended-Tekhex

Everything in hexadecimal representation.

<Segment record> <Public absolutes> <Local absolutes> <Code> <End>

<Segment record> (repeatable) contains two records :
 <Public relative symbols> <Local relative symbols>

 <Public relative symbols> (repeatable) :
 %<Block length>3<CHK> <Segment definition>
 <Symbol definition> <eoln>

 <Local relative symbols> (repeatable) :
 %<Block length>3<CHK> <Segment definition>
 <Symbol definition> <eoln>

<Public absolutes> (repeatable) :

```

    %<Block length>3<CHK>e%_ABSOLUTE_VAR00<x>0<x>
>
    <Symbol definition> <eoln>

    <Local absolutes> (repeatable) :
    %<Block length>3<CHK>e%_ABSOLUTE_VAR00<x>0<x>
>
    <Symbol definition> <eoln>

    <Code> (repeatable) :
    %<Block length>6<CHK> <Load address> <Data> <eoln>

    <End> :
    %<Block length>6<CHK> <Program entry> <eoln>

```

Explanation of records

<Block length>:

This record is not repeatable and contains :
Length of block, excluding '%' and <eoln>.

<CHK>:

Checksum.

<Segment definition>:

This record is not repeatable and contains :

1. Segment name length. One byte.
2. The name, No more than 16 characters.
3. '0'.
4. Base address length. One byte.
5. Base address (hexadecimal).
6. Length of segment length. One byte.
7. Segment length (hexadecimal).

<Symbol definition>:

This record is repeatable and contains :

1. Symbol type. One byte (1-8).
2. Symbol name length (hex). One byte.
3. Symbol name.
4. Value length (hex). One byte.
5. Value (hex).

This record is not repeatable and contains :
A number of zeroes (1-8), depending on CPU.

This record is not repeatable and contains :

1. Load address length. One byte.
2. Load address.

This record is repeatable and contains :
Data bytes in hex representation.

This record is not repeatable and contains:

1. Program entry length. One byte.
2. Program entry (hex).

For explanations see the HP manual: "Hosted Development System", Sections "Absolute file format" and "Linker symbol file format". (Review HP-documentation Hosted Development Systems Sections File Formats and Symbol File Formats for details about the HP-formats).

```

:nnoooo00dddddcc      Data record
      (nn=01 to 10, dd=data, cc=checksum, oooo=offset/address)

:01000002sssscc INTEL-EXTENDED: 8086 type segment record
      (ssss=segment)

:01000003ssssooooocc INTEL-EXTENDED: 8086 type program
entry record

:00oooo01FF           End of file record (Last)
      (oooo=program entry point address)

```

A typical file contains two EOF records. One is :00000001FF and the other is the program entry record (if it exists). The -Y linker option can be used to change the EOF records. -Y0 causes the linker to always end the Hex file with the :00000001FF record. -Y1 causes the linker to end the Hex file with the program entry record if it exists; otherwise the Hex file is ended with :00000001FF

"cc" is the checksum. Please note that the sum of all bytes (MOD 256), including the checksum itself, should equal zero.

MILLENNIUM

/aaaannccddddddee Data record (nn=01 to 10)

/00000000 End of file record (Last)

aaaa	Load address
nn	The number of data bytes
cc	Checksum, from adding digits a and n MOD 256
dd	Data bytes
ee	Checksum, formed by adding all digits d MOD 256

MOTOROLA ("S-records")

S0nnppppxxxxxxxxcc First record in file (address pppp=0000)
(with the name of the first module)

S1nnppppddddddcc Data record with a 16-bit address

S2nnppppppddddddcc Data record with a 24-bit address

S8nnneeeeecc End of file record (nn=04), 24-bit
program entry

S9nnneeeeecc End of file record (nn=03), 16-bit
program entry

Explanation:

nn	Number of bytes (xx+pp+dd+cc) in record
pppp, pppppp	Load addresses
dd	Data bytes, 1 to 16.
xxxxxxx	The name of the first module (coded with two hex digits for every character)
eeee, eeeee	The program entry (execution) addresses
cc	Checksum is calculated with following formula:

$$cc := (255 - (\text{SUM}(pp) + \text{SUM}(dd) + \text{SUM}(xx) + \text{SUM}(ee) + nn)) \text{ MOD } 256.$$
MPDS-CODE

This is a binary "dump" of the linked program image. Code is dumped in sequential order from address zero to the highest address. "Holes" in the code image (i.e., uninitialized addresses) are padded with 00s by default. The -Y0 linker option can be used to cause the linker to pad these holes with FFs.

MPDS-SYMB

Each module has a Module record:

FE <Module name length> <Module name>
 <Record length> <Constant> <Symbol record>

The Symbol record consists of a variable number of records:

<Symbol name length> <Symbol name> <Symbol value>

Module name length and Symbol name length are one byte long.

Module name and Symbol name are length long.

Record length (three bytes) is the length of the remaining block.

Constant is 2 or 3, depending on Symbol value length (2 or 3 bytes).

Symbol value is two or three bytes long.

MSD

This format contains only symbol information.

<Value> <Symbol name> <End of line>

<Value> is two bytes (hex).

<Symbol> name is global entry or local name.

This format can be modified by using the -Y linker option as follows:

- Y0 enables all symbols to be module:symbolname
- Y1 enables local and line symbols to be module:symbolname
- Y2 enable line symbols to be module:linenumber

NEC-SYMBOLIC

Symbol Table Start	2 char		4 char	max 8 char
	#	04		
	;	FF	4 Blanks (spaces)	Module Name
		Type	Symbol Value	Public Sym Name
		Type	Symbol Value	Public Sym Name
		.	.	.
		.	.	.
	<	Type	Symbol Value	Local Sym Name
		Type	Symbol Value	Local Sym Name
		.	.	.
		.	.	.
	;	FF	4 Blanks (blanks)	Module Name
		Type	Symbol Value	Public Sym Name
	=			

NOTE: Each row ends with carriage return and line feed.

Type	Attribute
Value	Number
00	Code
01	Data
02	Bit
05	Module Name
FF	

PENTICA-A, -B, -C, -D

<File Header> <End of line>
 <Symbol declaration> <End of line>
 <File Trailer> <End of line>

Explanation:

<File Header>

File header: The word "BEGIN" in upper case letters.

<Symbol declaration>

Consists of:

- a:
 1. Name of the symbol. No set amount of bytes.
 2. The letter 'P'.
 3. Value of entry or local (in hex).
- b:
 1. Value of entry or local (in hex followed by H).
 2. Name of the symbol. No set amount of bytes.
- c:
 1. Name of the symbol. No set amount of bytes.
 2. Value of entry or local (in hex).
- d:
 1. Value of entry or local (in hex).
 2. The letter 'X'.
 3. Name of the symbol. No set amount of bytes.

<File Trailer>

File trailer. The word "END" in uppercase letters.

The linker option -Y can be used to modify this format as follows:

- Y0 enables all symbols to be module:symbolname
- Y1 enables local and line symbols to be module:symbolname
- Y2 enable line symbols to be module:linenumber

Note

Three bytes addresses are used to represent addresses larger than two bytes.

RCA

48 null characters	Header
!Maaaa dddddddddddd;	First record
aaaa dddddddd;	Data record (< = 16 bytes)
aaaa dddddddd	Last data record (< = 16 bytes)
48 null characters	Trailer
aaaa	Load address.
dd	Data bytes.

SYMBOLIC, TYPED

<I>0<SYMBOL><CHK>	Start of a new module.
<I>1<LA><DATA><CHK>	Absolute data record in current module.
<I>2<LA><DATA><CHK>	Relocatable data record in current module.
<I>3<VAL><SYMBOL><TYPE><CHK>	Absolute local in current module.
<I>4<VAL><SYMBOL><TYPE><CHK>	Relocatable local in current module.
<I>5<VAL><SYMBOL><TYPE><CHK>	Absolute global in current module.
<I>6<VAL><SYMBOL><TYPE><CHK>	Relocatable global in current module.
<I>7<VAL><CHK>	Absolute program entry in current module.
<I>8<VAL><CHK>	Reloc. program entry in current module.
<I>3<CHK>	Last record in file

NOTE: <TYPE> is only available in the format TYPED.

Record	Description
<I>	A one character identifier.% = TYPED and # = SYMBOLIC.
<LA> <VAL>	A load address or value with 4 possible variations, where the first character in the record gives the type. 16-bit values are expressed as 4 hex ASCII digits and 32-bit values as 8 hex ASCII digits For other processors than 8086, only type 0 and 1 are used. 0 <16-bit address> 1 <32-bit address> 2 <16-bit 8086-type para. no.> <16-bit offset> 3 <16-bit 8086-type para. no.> <32-bit offset>
<DATA>	A variable length record, consisting of a 2-digit hex ASCII value holding the number of data bytes in the record. After the length indicator comes data bytes each byte represented by a 2-digit hex ASCII number. An exception is relocatable 8086-type paragraph numbers, which are 16-bit words written as: \$ <4 hex ASCII digits>.
<SYMBOL>	A variable length record, consisting of a 2-digit hex ASCII value holding the length of the symbol, followed by the symbol itself. All eventual non-printable characters are converted to "." (dot).
<TYPE>	A variable length record, consisting of a 2-digit hex ASCII value holding the number of bytes in the variable part of the type record. After the length indicator comes the type information each byte expressed as a 2-digit hex ASCII number.
<CHK>	A 2-digit hex ASCII value containing the modulus 256 sum of all characters in the record (excluding <CHK>). After the <CHK> record, an end of line character is output.

TI7000 (TMS7000)

K0000xxxxxxx7ccccF

Header with
name (xxxxxxxx)

3aaaa*ddBdddddBdddddBddddd7ccccF

Data record (1-16 bytes)

1aaaa7ccccF

Program entry record
End of file record.

:

3aaaa

Load address

Bdddd

Data word

*dd

Data byte

7cccc

Checksum

F

End of line

The checksum (cccc) is the negated sum of all characters in the line up to and including the checksum tag (7).

ZAX

\$\$<module name>

•

```
'<entry name> <hex address or value>H
```

•

```
'<local name> <hex address or value>H
```

•

```
' '#<line number> <hex address>H
```

•

The Time Saver

Librarian



TM

ARCHIMEDES
SOFTWARE

LIBRARIAN

1. Introduction to the XLIB Librarian

1.1 Library Basics

As a preface to the discussion of the XLIB Librarian itself, this section reviews the concept of library files and how they are used in both C-6811 and Assembler programs.

1.1.1 Library Files

Before we discuss the XLIB Librarian itself, we need to take a brief look at libraries: what they are, how they are created and used, and, typically, who uses libraries and for what purpose.

In the terms of the Archimedes 68HC11 C Compiler, Assembler, Linker and Librarian, a library is simply a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed.

Normally, the modules in a library file all have the "Library" attribute, which means that they will only be loaded by the Linker if they are actually needed in the program. This is referred to as "demand loading" of modules. Refer to the linker chapter, "More About How XLINK Works" for more complete description of the linking process.

On the other hand, a module with the "Program" attribute is ALWAYS loaded when the file in which it is contained is processed by the Linker.

Please note that a library file is really no different from any other relocatable object file (.R07 filetype) that comes out of the A6801 Assembler or C-6811 Compiler, except that it would include a number of modules of the Library type.

1.1.2 C-6811 Programs and Libraries

All C-6811 programs make use of libraries -- a number of standard library files are distributed with the C-6811 Compiler Kit. You must select one of these files to use in your programs according to which C-6811 "memory model" you have selected (e.g., for the large memory model a file named CL6811.R07 is used).

These libraries include "helper" functions that are required by C-6811 programs, as well as the standard C-callable functions (printf, malloc, strcpy, etc.). Each standard C-6811 library contains over 100 modules, each of which can be loaded independently of the others by the Linker as it is needed by the program.

Most C-6811 users will use the XLIB Librarian at some point, for one of the following reasons:

- You will usually need to replace or modify a module in one of the standard C-6811 libraries. For example, the Librarian is used to replace the distribution versions of the CSTARTUP and/or PUTCHAR modules with ones that you have customized. This is part of configuring the library for your application. (Also see the -A and -C Linker options described in the "XLINK Command Detail" section of the Linker chapter which can be used to alter the way that Library and Program modules are loaded).
- You can add your own C or assembler modules to the standard C-6811 library file so they will always be available whenever you link a C program.
- You can create your own custom library files that can be linked into your programs, as needed, along with the standard C-6811 library (you can list as many "library" files in a linkage as you desire as XLINK treats them just like any other input file).

NOTE

The C-6811 Compiler creates modules with a Program attribute by default. The "-b" Compiler option is used to specify a Library-type output module.

NOTE for C-6811 Users

Please refer to the C-6811 Compiler chapter in this manual for specific directions on choosing the correct standard library and configuring it to meet the requirements of your application.

1.1.3 Assembler Programs and Libraries

Assembler-only users are not required to use libraries. However, there are advantages to using libraries, especially when writing medium and large-size assembler applications:

- If you write "utility" modules which are used in more than one project, you can combine the modules into a single library file. This simplifies the linking process as you do not need to include a long list of input files for all the needed modules, but only specify the one library file. Only the Library module(s) needed for the program will be included in the output file.
- Program maintenance is simplified as you can place multiple modules in a single assembler source file. Each of the modules can be loaded independently as a Library module.
- The number of object files that make up an application is reduced, which simplifies maintenance and documentation.

You can create your own assembly-language library files using one of two basic methods:

- A library file can be created by assembling a single assembler source file which contains multiple Library-type modules. See the "Multi-Module Files and Libraries" section in the Assembler chapter for more information on how this is done. The resulting library file can then be modified with the use of XLIB.
- A library file can be produced by using the XLIB Librarian to merge any number of existing modules together to form a user-created library.

NOTE

The NAME and MODULE assembler directives are used to declare modules as being Program or Library type, respectively. See Modules and Module Directives in the Assembler chapter for more information.

1.2. XLIB Librarian Features

The Archimedes XLIB Librarian is a program which enables you to manipulate relocatable object files (default filetype .R07) and the modules they contain. Just as a text editor allows you to manipulate source code files, the XLIB Librarian allows you to manipulate relocatable object code files.

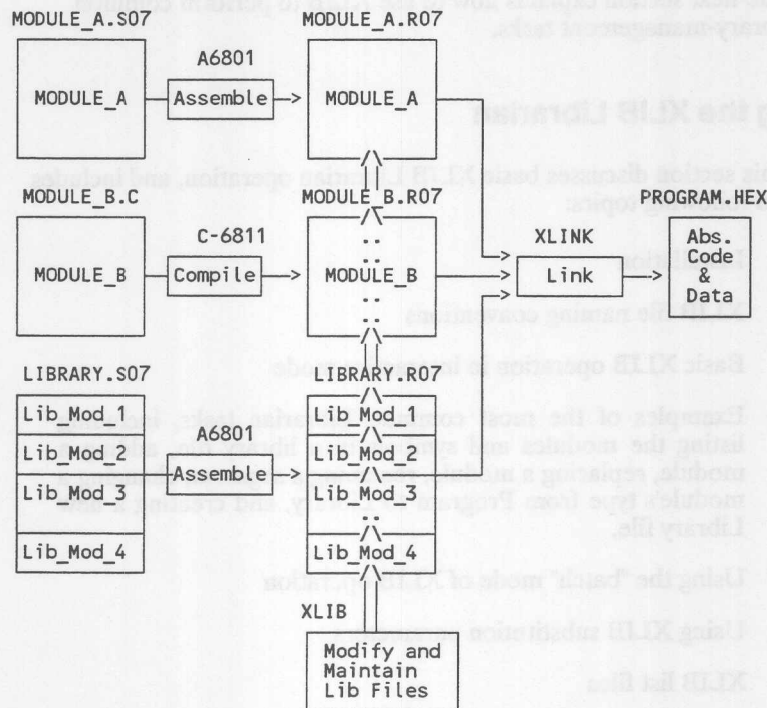
The XLIB Librarian offers you the following features:

- Support for modular programming
- Modules can be listed, added, inserted, replaced, deleted or renamed
- Segments can be listed and renamed
- Symbols listed and renamed
- A module can be changed from Program to Library type, and vice-versa
- Interactive or batch mode operation
- Full set of library listing operations
- Command to display a directory of relevant files on disk
- On-line help command

1.3 XLIB in the Development Cycle

The XLIB Librarian manages relocatable object files (.R07 default filetype) which have been assembled by the Archimedes A6801 Assembler or compiled by the Archimedes C-6811 Compiler.

These modules are formatted in the proprietary UBROF format generated by the Archimedes Assembler and Compiler, and required by the Archimedes XLINK Linker (see Glossary: UBROF).



In the above diagram, the source files MODULE_A.S07 (assembly-language) and MODULE_B.C (C-language) are single-module source programs. LIBRARY.S07 is a multi-module assembly-language source file.

The relocatable files `MODULE_A.R07` and `MODULE_B.R07` are not considered "libraries" because they only contain a single module each. On the other hand, file `LIBRARY.R07` contains four (presumably Library-type modules) and so is considered a "library" file.

Using the XLIB Librarian, you could manage the modules in the file `LIBRARY.R07`. For example, you could add `MODULE_B.R07` to the library file instead of including it in the link as a separate file.

The next section explains how to use XLIB to perform common library-management tasks.

2. Using the XLIB Librarian

This section discusses basic XLIB Librarian operation, and includes the following topics:

- Installation
- XLIB file naming conventions
- Basic XLIB operation in interactive mode
- Examples of the most common Librarian tasks, including listing the modules and symbols in a library file, adding a module, replacing a module, renaming a segment, changing a module's type from Program to Library, and creating a new Library file.
- Using the "batch" mode of XLIB operation
- Using XLIB substitution parameters
- XLIB list files

This section includes examples of only the most commonly-used XLIB commands -- Next section contains a complete reference to all commands and parameters.

2.1 XLIB Installation

Before proceeding with this Chapter, you should install the XLIB Librarian (i.e., the XLIB.EXE file) on your computer system.

IMPORTANT

If you have not already done so, please install the XLIB software at this time on your computer as directed in Appendix-A: Installation.

XLIB supports a number of DOS "environment variables" that are used to specify default command line parameters, which can simplify Librarian operation. These variables are described in the "XLIB Environment Variables" section in this chapter.

2.2 XLIB File Naming Conventions

- .R07 The default filetype for library files (which are essentially the same as other relocatable object files)
- .LST Default filetype for XLIB list files
- .XLB Default filetype for XLIB "command" files, which are used in "batch" mode (see the "Librarian Batch Mode")

These defaults may be overridden by supplying an explicit filetype when specifying a file.

2.3 Basic XLIB Operation

2.3.1 Running XLIB

There are two ways in which the XLIB Librarian can be operated: interactive mode and batch mode. To invoke the Librarian in interactive mode, you simply type "XLIB" by itself:

```
XLIB
```

To invoke the Librarian in batch mode, you specify the name of a "command" file (default filetype of .XLB) and optional parameters on the command line:

```
XLIB command file [p0,p1,...]
```

The interactive mode of operation is the simpler method for occasional use, and will be used in this Chapter for all examples.

See the "Librarian Batch Mode" section for using the more advanced (and powerful) batch mode of operation, which allows multiple Librarian tasks to be performed with a single XLIB command line.

2.3.2 The XLIB Interactive Mode

As we have mentioned, the simplest way to operate XLIB is in interactive mode, where XLIB prompts you for all commands and parameters, as in the following XLIB session (user input is underlined>):

```
C>XLIB
```

```
Archimedes Universal Librarian V3.16/DXT
(c) Copyright Archimedes Software Inc. 1991
```

```
*D-C 68HC11      <-- Must start by defining CPU type
*H              <-- XLIB help command

help            display-options    exit
quit            remark            list-object-code
list-all-symbols list-crc        list-modules
list-entries    list-externals    list-segments
list-date-stamps rename-module    rename-entry
rename-external rename-global    rename-segment
delete-modules  insert-modules    replace-modules
fetch-modules   make-library      make-program
compact-file    define-cpu        echo-input
on-error-exit   directory
```

In the above example, at the XLIB "*" prompt, the "D-C" command is entered first (this is a short form of the DEFINE-CPU command). This command MUST be entered at the beginning of every XLIB session (the Linker will issue an error message if you forget).

Next the "H" (short form of HELP) command is used to display the complete list of all the available XLIB command. We will be demonstrating some of these commands in this Chapter; all of them are described in the "Librarian Command Reference" section.

At the XLIB "*" prompt, you can enter any of the above commands including all of the necessary parameters to complete the command, all on one line.

You may also type just the name of the command and XLIB will prompt you for the necessary parameters one at a time. Using this mode of operation, you do not need to remember the order of parameters for each command.

Both of these modes of operation will be shown in the next section, where we will illustrate the use of the most common Librarian commands. (Refer to the "XLIB Command Reference" section for a complete list of XLIB commands and parameters.)

2.3.3 Common Librarian Tasks

Listing the Modules in a Library

You can use the LIST-MODULES (L-M) command to display a list of the modules in a library file:

```
*L-M
Object file = CL6811
List file =
Start module =
End module =
1. Pgm  CSTARTUP
2. Lib  exit
3. Lib  putchar
4. Lib  abort
.
164. Lib  ?DL_L_SHIFT_L10
165. Lib  ?DL_R_SHIFT_SPEC_L10
166. Lib  ?DL_R_SHIFT_L10
.
```

In the above example, the Object file refers to the library file that we wish to list the contents of (modules). In this case, we specified one

of the standard C-6811 Library files, CL6811.R07 (a .R07 filetype is assumed, so we do not need to specify it).

In the resulting list, the internal number (1 - 166) and type (Pgm for Program, or Lib for Library) is shown for each module in the CL6811.R07 library file.

A List file can be specified as an optional parameter, in which case the requested list is sent to the specified file (using a default filetype of .LST). Use the device name "PRN" to send the file to the printer.

The Start module and End module optional parameters set the beginning and ending modules to include in the list. Either the module's name or its internal number may be specified here.

NOTE

*All module, segment and symbol names are case-sensitive.
File names are not case-sensitive.*

The same results could have been obtained by entering the following command at the XLIB prompt:

```
*L-M CL6811
```

Listing the Symbols in a Library

The LIST-ALL-SYMBOLS (L-A-S) command is used to display a list of the segments, entries (i.e. public/global symbols) and external references for a range of modules in a library file:

```
*L-A-S
Object file = CL6811
List file =
Start module = 160
End module = 166

160. Lib ?FLT TO DBL_L10
      Rel RCODE
      Ent ?FLT TO DBL_L10
      Ext ?DL_R_SHIFT_L10
161. Lib ?DBL TO FLT_L10
      Rel RCODE
      Ent ?DBL TO FLT_L10
      Ext ?DL_L_SHIFT_L10
162. Lib ?DBL TO LONG_L10
      Rel RCODE
      Ent ?DBL TO SL_L10
      Ent ?DBL TO UL_L10
      Ext ?DL_R_SHIFT_L10
      Ext ?DL_L_SHIFT_L10
      .
166. Lib ?DL_R_SHIFT_L10
      Rel RCODE
      Ent ?DL_R_SHIFT_L10
```

In the above example, we have listed the symbols in the CL6811.R07 file for the modules 160 - 166. Listed for each module is a list of the segments, entries (global/public symbols) and external symbol references.

The same results could have been obtained using the following command (note that the double commas ",," indicates the List file parameter is not specified):

```
*L-A-S CL6811,,160 166
```

Other related XLIB commands include (see the "XLIB Command Reference" section for details):

LIST-ENTRIES (L-EN): list global/public symbols

LIST-EXTERNALS (L-EX): list external references

LIST-SEGMENTS (L-S): list segments

LIST-DATE-STAMPS (L-D-S): list generation date of modules

Updating Modules in a Library

When you modify the source code for a module which is in a library file, and re-assemble or re-compile it, you can update the object module in the library using the REPLACE-MODULES (REP-M) command.

The following example illustrates how this command could be used to update the C-6811 Library file (CL6811.R07) with the modules contained in the file CINTER.S07:

```
*REP-M
Source file = CINTER
Destination file = CL6811

Replacing module '?SEG_INIT_L17' <-- names of modules being replaced
Replacing module '?CALLING_YOU_L17'
.
.
Replacing module '?VERSION_L17'
```

In the previous example, the Source file (CINTER.R07) is the object file that contains the new module(s). The Destination file is the library file that contains the old module(s). This command copies all of the modules that it finds in file CINTER.R07 to the modules of the same name in file CL6811.R07. In other words, the modules in CINTER.R07 are used to update the library file.

XLIB updates the library (Destination) file with all the modules it finds in the Source file.

The following command performs the same function as above:

```
*REP-M CINTER CL6811
```

Adding a Module to a Library

Creating a New Library File

A module can be added to a library file, or a new library can be created, with the use of the **FETCH-MODULES (F-M)** command. This command takes module(s) from the specified Source file (.R07 filetype) and places them in the specified Destination file (the library):

```
*F-M
Source file = EXAMPLE      <---- file with module(s) to add to...
Destination file = NEWLIB  <---- library file
Start module =
End module =
```

The above command adds the module(s) from file **EXAMPLE.R07** to the library file **NEWLIB**. If the file **NEWLIB** does not already exist, **FETCH-MODULES** will create a new file with that name and add the modules from **EXAMPLE** to it.

Other related **XLIB** commands include (see the "XLIB Command Reference" section for details):

INSERT-MODULES (I-M): change order of modules in library

DELETE-MODULES (D-M): delete modules from a library

Changing Names in a Library

The names of various items within a library can be changed without having to re-assemble or re-compile the source code. For example, the **RENAME-SEGMENTS (R-S)** command can be used to change the name of a segment:

```
*R-S
Object file = EXAMPLE
Old name = UDATA
New name = NEWSEG
Start module =
End module =
```

1 substitution(s) made.

In this example, segment **UDATA** is renamed to **NEWSEG** throughout the module **EXAMPLE** (a range of modules within the file can also be specified).

Other related XLIB commands include (see the "XLIB Command Reference" section for details):

RENAME-MODULE (REN-M): rename a module

RENAME-ENTRY (R-EN): rename public symbols in a module

RENAME-EXTERNAL (R-EX): rename an external reference

RENAME-GLOBAL (R-G): rename both public and externals

Changing a Module's Type

The MAKE-LIBRARY (M-L) command can be used to change a Program module to a Library module:

```
*M-L
Object file = EXAMPLE
Start module =
End module =
```

The above command would change the module(s) found in EXAMPLE.S07 from Program type to Library type. This command is useful for setting C-6811 modules to Library type so that they can be used in a library (by default, modules created with the C-6811 Compiler have a Program type).

The MAKE-PROGRAM (M-P) command is used to change a module's attribute from Library to Program.

Quitting XLIB

The QUIT (Q) or EXIT (EX) commands can be used to terminate an XLIB session.

2.4 Librarian Batch Mode

This section describes the more advanced "batch" mode of Librarian operation. In batch mode, multiple XLIB commands can be placed in a command file along with optional "substitution" parameters (described below) that are filled in with command line parameters when XLIB executes.

The batch mode of XLIB operation is invoked as follows:

```
XLIB command file [p0,p1,...p9]
```

... where command file specifies the name of an ASCII text file which contains XLIB commands. The default filetype for XLIB command files is ".XLB" if one is not specified.

For example, a simple XLIB command file "DOLIB.XLB" might contain the following commands to update the modules in library file MYLIB with the modules from file MYPROG:

```
DEFINE-CPU 68HC11  
REPLACE-MODULE MYPROG MYLIB  
EXIT
```

The above XLIB session could then be executed as follows:

```
XLIB DOLIB
```

The optional command line parameters p0 through p9 can be used to pass values to corresponding substitution parameters in the command file.

Substitution Parameters

XLIB supports 10 substitution parameters, designated as "\0" through "\9", which can be used in XLIB command files. These substitution parameters are replaced by the corresponding command line parameters (which you specify on the XLIB command line) when the command file is executed.

For example, assume the "DOLIB.XLB" command file contains the following lines:

```
DEFINE-CPU 68HC11
REPLACE-MODULE \0 \1
EXIT
```

If this file is executed with the following command line:

```
XLIB DOLIB FILE1 PROJLIB
```

... it would result in the following XLIB commands being executed:

```
DEFINE-CPU 68HC11
REPLACE-MODULE FILE1 PROJLIB
EXIT
```

Substitution Parameter Defaults

When you use substitution parameters in an XLIB command file, you may specify a default value for XLIB to use if that parameter is not specified on the command line which invoked XLIB. A default value is given by enclosing it in single quotation marks (') immediately following the substitution parameter.

For example, consider the following line placed in an XLIB command file named "CFILE.XLB":

```
REP-MOD MYLIB,,\0,\1'QRS'
```

Then, if the following XLIB command line was entered:

```
XLIB CFILE,ABC
```

... the following XLIB command would be produced:

```
REP-MOD MYLIB,,ABC,QRS
```

Note that the value of the second substitution parameter (\1) is not defined for the command line shown (since only one parameter following the command file name was specified). Therefore, the default value for "\1" of 'QRS' is used.

The Interactive Parameter

The XLIB batch mode also provides a powerful interactive parameter. If you place the special parameter "\" in the XLIB command file, when XLIB encounters it, it will stop and prompt the user for a value for that parameter. A prompt string can follow the "\" parameter by enclosing it in single quotes (').

For example, the following line could be placed in an XLIB command file to prompt the user for a file to list:

```
LIST-MOD \?'ENTER NAME OF FILE TO LIST:' PRN
```

The prompt "ENTER NAME OF FILE TO LIST:" will be displayed, and the filename entered by the operator will be used as a parameter for the LIST-MODULES command. The output list will be sent to the PRN device.

2.5 Librarian List Files

Many of the XLIB commands can generate a listing which contains information about the modules and symbols in the specified library. By default, this listing is sent to the screen, but can be directed to a disk file by providing the name of the file as the List file parameter in the command (see the "XLIB Command Reference" section for details on which XLIB commands can cause a List file to be generated.)

If the listing is sent to a disk file, or to the PRN (printer) device, there will be a header placed at the beginning of the file, with the following information:

.		
#	Command	= list-modules
#	Object file	= progs.R07
#	List file	= progs.lst
#	Start module	= Default parameter
#	End module	= Default parameter
.		
.		

A listing of symbols may also be included in the list file, depending on the XLIB command. The following table lists the prefixes that XLIB uses to tag the various types of symbols that can appear in a listing file:

nn. Pgm	<symbol>	program module in position nn
nn. Lib	<symbol>	library module in position nn
Ext	<symbol>	an external symbol
Ent	<symbol>	a public symbol
Loc	<symbol>	a local symbol
Rel	<symbol>	an RSEG segment
Stk	<symbol>	a STACK segment
Com	<symbol>	a COMMON segment

3. XLIB Command Reference

This Chapter consists of a detailed reference for all XLIB Librarian commands and associated parameters. The topics in this Chapter include:

- Summary table of XLIB commands grouped by function
- Summary of XLIB MS-DOS "environment variables"
- Notes on the XLIB command syntax
- Default XLIB command parameters
- Detailed reference for all XLIB commands:

help	display-options	exit
quit	remark	list-object-code
list-all-symbols	list-crc	list-modules
list-entries	list-externals	list-segments
list-date-stamps	rename-module	rename-entry
rename-external	rename-global	rename-segment
delete-modules	insert-modules	replace-modules
fetch-modules	make-library	make-program
compact-file	define-cpu	echo-input
on-error-exit	directory	

3.1 XLIB Command Summary

The following table summarizes the XLIB commands, which are grouped by command functions.

Library Listing Commands

LIST-ALL-SYMBOLS	List every symbol in modules
LIST-CRC	List CRC values of modules
LIST-DATE-STAMPS	List dates of modules
LIST-ENTRIES	List PUBLIC symbols in modules
LIST-EXTERNALS	List EXTERN symbols in modules
LIST-MODULES	List modules in relocatable file
LIST-OBJECT-CODE	List low level relocatable code
LIST-SEGMENTS	List segments in modules

Library Editing Commands

DELETE-MODULES	Remove modules from library
FETCH-MODULES	Add modules to a library
INSERT-MODULES	Move modules around in a library
MAKE-LIBRARY	Change module to "library" type
MAKE-PROGRAM	Change module to "program" type
RENAME-ENTRY	Rename PUBLIC symbols
RENAME-EXTERNAL	Rename EXTERN symbols
RENAME-GLOBAL	Rename EXTERNs and PUBLICs
RENAME-MODULE	Rename one or more modules
RENAME-SEGMENT	Rename one or more segments
REPLACE-MODULES	Update executable code

Miscellaneous Library Commands

COMPACT-FILE	Shrinks library file size
DEFINE-CPU	Specifies CPU type
DIRECTORY	Displays available object files
DISPLAY-OPTIONS	Displays XLIB options
ECHO-INPUT	Command file diagnostic tool
EXIT	Return to operating system
HELP	Display help information
ON-ERROR-EXIT	Quit on batch error
QUIT	Return to operating system
REMARK	Comment in command file

3.2 XLIB Environment Variables

XLIB supports a number of DOS environment variables (see Glossary) which can be defined in the PC/MS-DOS host environment using the DOS "SET" command. These variables can be used to create defaults for various XLIB options so they do not have to be specified on the command line.

To make these settings automatic, you can place the SET commands in your system's AUTOEXEC.BAT file (or in your "login script" if you are running on a network).

XLIB_PAGE

Sets the number of lines per page (10 - 100) for XLIB listing files. The default is a non-paged list.

Example: SET XLIB_PAGE=66

XLIB_COLUMNS

Sets the number of columns per line (80 - 120) for listings. The default is 80 columns.

Example: SET XLIB_COLUMNS=132

XLIB_CPU

Sets the CPU type. If this variable is set to "68HC11", the DEFINE-CPU command does not need to be entered at the beginning of an XLIB session.

Example: SET XLIB_CPU=68HC11

XLIB_SCROLL_BREAK

By setting this variable to nnn lines (16 - 100), XLIB will pause and wait for the <Enter> key to be pressed after nnn lines on the screen have scrolled by.

Example: SET XLIB_SCROLL_BREAK=22

3.3 XLIB Command Syntax

3.3.1 Command Delimiters

Spaces, tabs or commas (,) can be used as delimiters between parameters in an XLIB command.

Two commas entered in a row (,,) indicates that the parameter which would otherwise be specified between the commas is to be taken as the default value for that parameter.

3.3.2 Command Abbreviations

XLIB commands can be entered literally as listed, or can be abbreviated in any way that does not present a conflict as to which command is being requested.

The shortest allowable abbreviation for each XLIB command is given in the reference section which follows.

3.3.3 Command Defaults

Some XLIB commands accept optional parameters for a List file, a Start module and an End module. For the List file parameter, the default is to send the listing output to the screen. For the Start and End module parameters, the first and last modules in the file are used for defaults.

3.3.4 Specifying Modules

Many of the Librarian commands accept a module name as a parameter. However, there are several ways in which you can specify a module:

Name

The actual name of the module may be entered. The module name is assigned by the C-6811 Compiler for C program, or by the NAME or MODULE directive in an Assembler program (upper/lower case is significant for module names).

Number

Modules within a library file are numbered from 1 (the first module in the library) through the number of modules in the library. You can specify a module by the (decimal) number of its position within the library.

\$

The dollar sign can be used to specify the last module in the library.

Expression

You may specify a module as <module name> plus or minus an integer. For example, "INIT + 5" specifies the fifth module after the module named "INIT"; and "\$-1" specifies the next to the last module in the file.

3.4 XLIB Command Reference

This section consists of an alphabetical listing of all XLIB commands and parameters. The abbreviated form of each command is shown to the right of the full command name.

COMPACT-FILE

C-F

Syntax

```
C-F <object file>
```

This command concatenates short absolute records into longer records of variable length. This can speed up the linking process when large libraries are being linked.

Example

To pack RELFIL1, enter:

```
C-F RELFIL1
```

DEFINE-CPU

D-C

Syntax

```
D-C <CPU>
```

This command sets the CPU type for Librarian operations. Also see the "XLIB Environment Variables" section.

Example

To specify the 68HC11 for further operations, enter:

```
D-C 68HC11
```

DELETE-MODULES**D-M****Syntax**

```
D-M <object file> [<start module>] [<end module>]
```

This command removes the specified module(s) from the library file specified by <object file>. All modules from <start module> through <end module> are deleted.

Example

To remove just MOD1 from LIBFILE, enter:

```
D-M LIBFILE,MOD1,MOD1
```

DIRECTORY**DIR****Syntax**

```
DIR [<path>]
```

This command displays the object files available for the current CPU type (.R07 for the 68HC11). If the optional path is not specified, the current directory will be used.

Example

To show all object files in the PROJECT3 subdirectory, enter:

```
DIR \A6811\PROJECT3
```

DISPLAY-OPTIONS**D-O****Syntax**

```
D-O [<list file>]
```

This command displays all of the possible CPU types which can be specified for XLIB, along with the default relocatable file extension for each one.

Following the CPU types is a list of internal tags used by XLIB to classify the parts of relocatable files.

ECHO-INPUT**E-I****Syntax**

E-I

This command causes all lines from a command file to be shown on the screen as they are read and executed by XLIB in batch mode. This command is useful as a debugging tool when creating command files. It has no effect in interactive mode.

EXIT**EX****Syntax**

EX

This command causes XLIB to terminate and return to the operating system.

FETCH-MODULES**F-M****Syntax**

F-M <source file> <dest file>
[<start module>] [<end module>]

This command reads modules from <source file>, from <start module> through <end module>, and appends those modules to the end of <dest file> (a library file).

This command is used to add modules to an existing library, or to create a new library file. If <dest file> does not exist, it will be created.

Example

To take the single module contained in MOD1 and add it to the end of library file LIBFILE, enter:

F-M MOD1,LIBFILE

HELP**H****Syntax**

```
H [<command>] [<list file>]
```

If the optional command (abbreviated or not) is not specified, this lists all 29 of the XLIB Librarian commands in their long form.

If command is specified, a brief description of that command will be listed.

Example

```
H          (displays a list of all XLIB commands)
H L-A      (describes the LIST-ALL-SYMBOLS command)
```

INSERT-MODULES**I-M****Syntax**

```
I-M <object file> <start module> <end module>
    <Before/After> <dest module>
```

This command is used for moving modules around within an object file. The list of modules to be moved is given by <start module> through <end module>. The location to which they are to be moved is given by <dest module>.

The list will be placed in front of or following the <dest module> depending upon the <Before/After> parameter. This parameter may be abbreviated as "B" or "A".

Example

To cause the order of the modules to be changed so that MOD2 is the first module, followed by MOD3 and then MOD1, enter:

```
I-M LIBFILE,MOD2,MOD3,B,MOD1
```

LIST-ALL-SYMBOLS**L-A****Syntax**

```
L-A <object file> [<list file>]
      [<start module>] [<end module>]
```

This command lists all of the symbols in the modules from <start module> through <end module> in the library <object file>.

Each module name is displayed with all the symbols within that module. Each symbol is tagged by a code which represents its type:

nn.	Pgm	<symbol>	program module #nn
nn.	Lib	<symbol>	library module #nn
	Ext	<symbol>	EXTERN symbol
	Ent	<symbol>	PUBLIC symbol
	Loc	<symbol>	Local symbol
	Rel	<symbol>	RSEG segment
	Stk	<symbol>	STACK segment
	Com	<symbol>	COMMON segment

NOTE

The LIST-ALL-SYMBOLS command is the only way to list local symbols which have been included in the relocatable code by the Assembler's LOCSYM + directive or the C-6811 Compiler's -rn switch.

Example

To list all the symbols and modules in LIBFILE, enter:

```
L-A LIBFILE
```

LIST-CRC**L-C****Syntax**

```
L-C <object file> [<list file>]
      [<start module>] [<end module>]
```

This command lists specified module names with their and associated CRC (Cyclic Redundancy Check) values.

The CRC is an error-detections code generated by the Assembler or C Compiler. The XLINK Linker uses this code to detect if an object file has been corrupted.

Example

To display the CRC codes for all modules in LIBFILE, enter:

```
L-C LIBFILE
```

LIST-DATE-STAMPS**L-D****Syntax**

```
L-D <object file> [<list file>]
      [<start module>] [<end module>]
```

This command lists the modules in <object file>, from <start module> through <end module>, and displays the date/time stamp for each module (the module generation date/time). An optional <list file> can be created.

Example

To show all date stamps for all modules in LIBFILE:

```
L-D LIBFILE
```

LIST-ENTRIES**L-EN****Syntax**

```
L-EN <object file> [<list file>]  
      [<start module>] [<end module>]
```

This command lists the modules in <object file>, from <start module> through <end module>, and displays the public symbols for each module. An optional <list file> can be created.

Example

To list all of the public symbols in file LIBFILE, enter:

```
L-EN LIBFILE
```

LIST-EXTERNALS**L-EX****Syntax**

```
L-EX <object file> [<list file>]  
      [<start module>] [<end module>]
```

This command lists the modules in <object file>, from <start module> through <end module>, and displays the external references for each module. An optional <list file> can be created.

Example

To list all of the modules in LIBFILE and display the external (EXTERN) symbols in each module, enter:

```
L-EX LIBFILE
```

LIST-MODULES**L-M****Syntax**

```
L-M <object file> [<list file>]
      [<start module>] [<end module>]
```

This command lists the names of the modules in <object file>, from <start module> through <end module>. An optional <list file> can be created.

Example

To display a list of all the modules in LIBFILE, enter:

```
L-M LIBFILE
```

LIST-OBJECT-CODE**L-O****Syntax**

```
L-O <object file> [<list file>]
```

This command lists all of the relocatable object code in the specified file. This is intended as a low level diagnostic tool for checking the output from an assembler or compiler.

Example

To list the object code in LIBFILE, enter:

```
L-O LIBFILE
```

LIST-SEGMENTS**L-S****Syntax**

```
L-S <object file> [<list file>]
      [<start module>] [<end module>]
```

This command lists the modules in <object file>, from <start module> through <end module>, and displays the segments within each module. An optional <list file> can be created.

Example

To list all of the modules in LIBFILE with the segments contained in each, enter:

```
L-S LIBFILE
```

MAKE-LIBRARY**M-L****Syntax**

```
M-L <object file> [<start module>] [<end module>]
```

This command causes each of the modules from <start module> through <end module> in <object file> to given the "Library" attribute (that is, the module will only be loaded by the Linker if referenced by some other module).

Example

To change module MOD1 in file LIBFILE to a "Library" type module, enter:

```
M-L LIBFILE,MOD1,MOD1
```

MAKE-PROGRAM**M-P****Syntax**

```
M-P <object file> [<start module>] [<end module>]
```

This command causes each of the modules from <start module> through <end module> in <object file> to given the "Program" attribute (that is, it will be always loaded by the Linker).

Example

To change module MOD1 in file LIBFILE to a "Program" type module, enter:

```
M-P LIBFILE,MOD1,MOD1
```

ON-ERROR-EXIT**O****Syntax**

O

This command tells XLIB that it is to cease execution of batch work if it encounters any error. The default mode of operation is to continue execution after displaying an error message.

QUIT**Q****Syntax**

Q

This command causes XLIB to terminate and return to the operating system.

REMARK**REM****Syntax**

REM <any text>

This keyword allows you to place comments in an XLIB command file. Any text following this command is ignored by the Librarian.

RENAME-ENTRY**R-EN****Syntax**

R-EN <object file> <old name> <new name>
[<start module>] [<end module>]

This command causes all public symbols which have <old name> to be renamed to <new name> in modules <start module> through <end module> in <object file>.

Example

To rename the public symbol GRAPE to APPLE in all of the modules in LIBFILE, enter:

```
R-EN LIBFILE, GRAPE, APPLE
```

RENAME-EXTERNAL**R-EX****Syntax**

```
R-EX <object file> <old name> <new name>  
      [<start module>] [<end module>]
```

This command causes all external references which have <old name> to be renamed to <new name> in modules <start module> through <end module> in <object file>.

Example

To rename the external symbol CHERRY to BANANA in all of the modules in LIBFILE, enter:

```
R-EX LIBFILE, CHERRY, BANANA
```

RENAME-GLOBAL**R-G****Syntax**

```
R-G <object file> <old name> <new name>  
      [<start module>] [<end module>]
```

This command causes all public and external symbols which have the name <old name> to be renamed to <new name>. This is equivalent to executing both RENAME-ENTRY and RENAME-EXTERNAL with the same parameters.

Example

To rename all public and external symbols named RED to BLUE, enter:

```
R-G LIBFILE, RED, BLUE
```

RENAME-MODULE**REN-M****Syntax**

```
REN-M <object file> <old name> <new name>
```

This command allows you to change the name of a module from <old name> to <new name> in <object file>. If there is more than one occurrence of module <old name> in the Library file, the first matching module will be renamed.

Example

To rename MOD1 to MODB in LIBFILE, enter:

```
REN-M LIBFILE,MOD1,MODB
```

RENAME-SEGMENT**R-S****Syntax**

```
R-S <object file> <old name> <new name>  
    [<start module>] [<end module>]
```

This command allows you to rename segments in relocatable files. All segments named <old name> in <object file> from <start module> through <end module> will be renamed to <new name>.

Example

To rename the segment CODE to EXEC throughout all of the modules in LIBFILE, enter:

```
R-S LIBFILE,CODE,EXEC
```

REPLACE-MODULES**REP-M****Syntax**

REP-M <source file> <dest file>

This command causes all of the modules in <source file> to be copied to the modules of the same name in <dest file> (a library file). This command is used to update a library file with new versions of modules after they changed.

NOTE

All the modules contained in <dest file> whose module name is matched by a module in <source file> will be replaced by the module from <source file>.

Example

To update the library file LIBFILE with the module(s) in MODS1, enter:

REP-M MODS1,LIBFILE

The Time Saver

Appendices



TM

ARCHIMEDES
SOFTWARE

APPENDIX - A

Installation

This Appendix describes the installation of the C-6811 development kit on a PC/AT host system.

NOTE for VAX/Sun/HP-3000 Users

This Appendix applies only if you are using the IBM PC/AT-hosted version of the software. If you are using the DEC/VAX, DEC/Ultrix, Sun/Unix or HP-3000/Unix version of the software, please refer to the Software Release Notes included with the package for installation instructions.

The following subjects are covered in this Appendix:

A.1 System Requirements

A.1.1 Memory

We recommend that your PC/AT host system have a minimum of 570kB of usable RAM in order to run this software. The usable amount of memory is that which the MS-DOS "CHKDSK" command prints out as "Bytes Free". XLINK, especially, can require more memory if you are linking large programs with a large number of symbols. It is strongly recommended that you remove memory-resident software (e.g. programs such as SideKick, disk cache, etc..) in order to free as much memory as possible. Note that these memory resident programs may interfere with the operation of the Archimedes executable programs, resulting in unpredictable behavior.

NOTE

See the "Linker Errors and Host Memory Management" section in the Linker chapter for more information on saving memory with XLINK.

A.1.2 Operating System

The Cross-Compiler Kit software will run under MS-DOS/PC-DOS versions 2.11 or later.

For Unix, Ultrix and VMS version requirements, refer to the Release Notes that came with your package.

A.1.3 Disk Requirements

A hard disk is recommended for running this software. When installed, the full development kit take approximately 2MB of disk space.

A.2 Software Installation

A.2.1 Modify Your CONFIG.SYS File

In order to assure that this kit can open the necessary number of files simultaneously, you must allow for 20 or more open files in the FILES statement located in the CONFIG.SYS file (this file is located in the root directory of your boot drive):

```
FILES=20          <-- must be set to at least 20
```

If your boot drive (normally drive C: or drive A:) does not contain a CONFIG.SYS file, you must create one with the above FILES command in it. You can use a standard program or text editor to create or modify the CONFIG.SYS file.

If you do not have the required FILES statement, the compiler may be unable to compile programs where "include" files are nested or the Linker may be unable to complete a linkage.

A.2.2 Copying the Disks to Your System

Copying the disks to your system can be done automatically by using the Install utility provided on the distribution diskettes.

To use this install utility, place the Archimedes distribution diskette #1 in the A: drive, then enter:

```
A:<cr>
install <cr>
```

At this point, you can follow the instructions on the screen.

This utility allows you to load the C-6811 files in a single-directory format or a multi-directory format. The default directory names are as follows:

Directory	Contents
Single:	
\ARCH	All files
Multiple:	
\ARCH	Text files
\ARCH\BIN	Executables
\ARCH\LIB	Libraries
\ARCH\SOURCE	C and ASM source files
\ARCH\INCLUDE	"#include" files

These default names can be overridden during installation.

The Install utility makes the following changes to your AUTOEXEC.BAT file if the multiple directory format is used:

```
set C_INCLUDE=<path to #include file directory>
set XLINK_DFLDIR=<path to library files directory>
```

If such changes are made on your AUTOEXEC.BAT file, reboot your system before you proceed.

A.3.1 Installing the DOS files

The Compiler, Linker, and Librarian are simultaneously released in two versions:.EXE and .OS2. The .OS2 version accesses a maximum of 640KB of memory. This version is useful if the application is relatively small, or if the host computer is not equipped with extended memory. In order to use this version of the software, rename the .EXE files as .SAV, then rename the .OS2 files as .EXE.

The .EXE version is based on the DOS/16M system developed by Rational Systems, Inc., Natick MA. This version is capable of accessing up to 16MB of PC RAM configured as EXTENDED memory. To use this extended-memory version, you must have an AT class compatible computer with a minimum of 700K of extended memory.

Although PC "extended memory" hardware is fairly standard and a high degree of compatibility has been achieved, there are some PC "clones" that are not compatible with DOS/16M or for which DOS/16M requires special notice via an environment variable.

In addition, there may be software or device driver incompatibilities. For example, DOS/16M is compatible with the VDISK and RAMDISK DOS utilities. However, if your system includes a RAM disk created with a different utility, that disk's use of extended memory may not be detected by DOS/16M and can cause both programs to attempt to access the same memory space, resulting in system malfunction.

If you are having problems with the extended memory version, try rebooting your system without any TSRs (Terminate and Stay Resident) utilities.

To determine if your PC's extended memory is compatible with DOS/16M, run the supplied PMINFO utility by typing:

```
PMINFO <cr>
```

The PMINFO utility should report the amount of available extended memory on your system. If this utility shows that you have at least 700K of extended memory available to DOS/16M, you can use the .EXE version. If, on the other hand, it returns impossible numbers or crashes the PC, then your system is not configured to run under DOS/16M. In this case, you can use the environment variable DOS16M to configure your system correctly. The following table shows the different values for different systems.

Machine	Setting	Comment
NEC 98 Series	1	*
PS/2	2	Automatically set
80386	3	Automatically set
80386	INBOARD	* for Intel Inboard
80386 W/VCPI detected	11	Automatically set if VCPI is
Fujitsu FMR-60, -70	5	*
AT&T 6300+	6	*
80286	7	* older phoenix BIOS
80286	9/19	Automatically set
80286	10	* faster switching
alternative Zenith Z-24K	13	* older BIOS

* you must set DOS16M for these machines

A.3.2 DOS Error Return Values

The software programs of this kit return status information to DOS which can be tested in an MS-DOS batch file using the "IF ERRORLEVEL n" statement. The supported error codes are listed below:

A6801 Assembler

- 0 Assembly successful
- 2 There was an error in the assembly

XLINK Linker/C-6811 Compiler:

- 0 The link was successful
- 1 There were warnings
- 2 There was a non-fatal error
- 3 Fatal error detected (XLINK aborted)

NOTE

If the XLINK -w (ignore warnings) option is specified, it will return a 0 even if warnings occur during the link.

These return codes can be used to test for errors in a batch file, as the following example using the XLINK command illustrates:

```
.
XLINK -f TESTLNK
IF ERRORLEVEL 3 GOTO ERRORS
IF ERRORLEVEL 1 GOTO WARNINGS
ECHO The Link was successful!
.
.
:WARNINGS
ECHO The Link ended with a warning or non-fatal error...
.
:ERRORS
ECHO The Link ended with a fatal error...
.
```

For an additional example batch file that you can run, see the ADEMO1.BAT file included with on the distribution diskettes and described in the Assembler chapter.

NOTE

Please refer to your MS-DOS documentation for more information on the use of environment variables, error return codes and batch files.

APPENDIX - B

C Error and Warning Messages

The error messages produced by the C-compiler falls into six categories:

1. Command line errors
2. Compilation warning messages
3. Compilation error messages
4. Compilation fatal error messages
5. Memory overflow message
6. Compiler internal errors

Command line errors

Command line errors occur when the compiler is invoked with bad parameters. The most common situation is that a file could not be opened. The command line interpreter is very strict about duplicate, misspelled or missing command line switches. However, it produces messages pointing out the problem in detail.

Compilation warning messages

Compilation warning messages are produced when the compiler has found a construct which typically is due to a programming error or omission. This appendix lists all warning messages.

Compilation error messages

Compilation error messages are produced when the compiler has found a construct which clearly violates the C language rules. Note that the Archimedes C-compiler is more strict on compatibility issues than many other C-compilers. In particular pointers and integers are considered as incompatible when not explicitly casted. This appendix lists all error messages.

Compilation fatal errors

Compilation fatal error messages are produced when the compiler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the compilation is immediately terminated. This appendix lists all compilation error messages (some marked as fatal).

Memory overflow message

The Archimedes C-compiler is a memory-based compiler that in case of a system with a small primary memory or in case of very large source files may run out of memory. This is recognized by a special message:

```
* * * C O M P I L E R   O U T   O F   M E M O R Y * * *
```

```
Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the cure is either to add system memory or to split source files into smaller modules. However, with 570K RAM the compiler capacity is sufficient for all reasonably sized source files. If memory is on the limit please note that the -q -x and -P command line switches cause the compiler to use more memory.

Compiler internal errors

During compilation a number of internal consistency checks are performed and if any of these checks fail the compiler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to Archimedes technical support group.

The compiler returns status information to DOS which can be used in conjunction with IF ERRORLEVEL in .BAT files.

- | | |
|---|--|
| 0 | Compilation successful |
| 1 | There were warnings (returns 0 if -w switch is on) |
| 2 | There were errors |
| 3 | Fatal error detected |

B.1 C Error Messages

Error [0]: Invalid syntax

Compiler could not decode statement or declaration.

Error [1]: Too deep #include nesting (max is 10)

Fatal. Compiler limit for nesting of #include files exceeded. Recursive #include could be the reason.

Error [2]: Failed to open #include file 'name'

Fatal. Could not open #include file. File does not exist in specified directories (check -I prefixes and environment variable C_INCLUDE) or is disabled for reading.

Error [3]: Invalid #include file name

Fatal. #include file name must be written <file> or "file".

Error [4]: Unexpected end of file encountered

Fatal. End of file encountered within declaration, function definition or during macro expansion. Probable cause is bad () or {} nesting.

Error [5]: Too long source line (max is 512 characters); truncated

Source line length exceeds compiler limit.

Error [6]: Hexadecimal constant without digits

Prefix "0x" or "0X" of hexadecimal constant found without following hexadecimal digits.

Error [7]: Character constant larger than "long"

Character constant contains too many characters to fit in space of a long integer.

Error [8]: Invalid character encountered: '\xhh'; ignored

Character not included in 'C' character set was found.

Error [9]: Invalid floating point constant

Too large or invalid syntax of floating point constant. See ANSI standard for legal forms.

Error [10]: Invalid digits in octal constant

Non-octal digit in octal constant. Valid octal digits are: 0, 1, 2, 3, 4, 5, 6 and 7.

Error [11]: Missing delimiter in literal or character constant

No closing delimiter ' or " was found in character or literal constant.

Error [12]: String too long (max is 509)

Compiler limit for length of single or concatenated strings exceeded.

Error [13]: Argument to #define too long (max is 512)

Lines terminated by '\ ' resulted in too long #define line.

Error [14]: Too many formal parameters for #define (max is 127)

Fatal. Too many formal parameters in macro definition (#define directive).

Error [15]: ';' or ')' expected

Invalid syntax of function definition header or macro definition.

Error [16]: Identifier expected

Identifier is missing in declarator, "goto" statement or in pre-processor line.

Error [17]: Space or tab expected

Pre-processor arguments must be separated from the directive with tab or space characters.

Error [18]: Macro parameter 'name' redefined

A #defined symbol's formal parameter was repeated.

Error [19]: Unmatched #else, #endif or #elif

Fatal. Missing #if, #ifdef or #ifndef.

Error [20]: No such pre-processor command: 'name'
'#' was followed by an unknown identifier.

Error [21]: Unexpected token found in pre-processor line
Pre-processor line was not empty after the argument part was read.

Error [22]: Too many nested parameterized macros (max is 50)
Fatal. Pre-processor limit exceeded.

Error [23]: Too many active macro parameters (max is 256)
Fatal. Pre-processor limit exceeded.

Error [24]: Too deep macro nesting (max is 100)
Fatal. Pre-processor limit exceeded.

Error [25]: Macro 'name' called with too many parameters
Fatal. A parameterized #define macro was called with more arguments than declared.

Error [26]: Actual macro parameter too long (max is 512)
A single macro argument may not exceed the length of a source line.

Error [27]: Macro 'name' called with too few parameters
A parameterized #define macro was called with fewer arguments than declared.

Error [28]: Missing #endif
Fatal. End of file encountered during skipping of text after a false condition.

Error [29]: Type specifier expected
Type description missing. It could happen in struct, union, prototyped function definitions/declarations or in K&R function formal parameter declarations.

Error [30]: Identifier unexpected
Invalid identifier. It could be an identifier in a type name definition like:

```
sizeof ( int *ident );
```

or two consecutive identifiers.

Error [31]: Identifier 'name' redeclared
Redeclaration of declarator identifier.

Error [32]: Invalid declaration syntax

Undecodable declarator.

Error [33]: Unbalanced '(' or ')' in declarator

Parenthesis error in declarator.

Error [34]: Attribute(s) given for "void" type; ignored

Attribute "const" or "volatile" given with type specifier "void"; ignored.

Error [35]: Invalid declaration of "struct", "union" or "enum" type

"struct", "union" or "enum" was followed by invalid token(s).

Error [36]: Tag identifier 'name' redeclared

"struct", "union" or "enum" tag is already defined in the current scope.

Error [37]: Function 'name' declared within "struct" or "union"

Function declared as member of "struct" or "union".

Error [38]: Invalid width of field (max is nn)

Declared width of field exceeds the size of an integer.

Error [39]: ';' or ';' expected

Missing ';' or ';' at the end of declarator:

```
char c cc  
int i;
```

Error [40]: Array dimension outside of "unsigned" "int" bounds

Array dimension negative or larger than can be represented in an unsigned integer.

Error [41]: Member 'name' of "struct" or "union" redeclared

Member of "struct" or "union" redeclared.

Error [42]: Empty "struct" or "union"

Declaration of "struct" or "union" containing no members.

Error [43]: Object cannot be initialized

Initialization of "typedef" declarator or "struct" or "union" member.

Error [44]: ';' expected

A statement or declaration needs a terminating semicolon.

Error [45]: ']' expected

Bad array declaration or array expression.

Error [46]: ':' expected

Missing colon after "default", "case" label or in '?'-operator.

Error [47]: '(' expected

Probable cause is a misformed "for", "if" or "while" statement.

Error [48]: ')' expected

Probable cause is a misformed "for", "if" or "while" statement or expression.

Error [49]: ';' expected

Invalid declaration.

Error [50]: '{' expected

Invalid declaration or initializer.

Error [51]: '}' expected

Invalid declaration or initializer.

Error [52]: Too many local variables and formal parameters (max is 1024)

Fatal. Compiler limit exceeded.

Error [53]: Declarator too complex (max is 128 '(' and/or '*')

Declarator contained too many '(', ')' or '*'.

Error [54]: Invalid storage class

Invalid storage-class for the object specified.

Error [55]: Too deep block nesting (max is 50)

Fatal. Too deep {} nesting in function definition.

Error [56]: Array of functions

Array of functions. The valid form is array of pointers to functions:

```
int array [ 5 ] ();    /* Invalid */
```

```
int (*array [ 5 ]) (); /* Valid */
```

Error [57]: Missing array dimension specifier

Multi-dimensional array declarator without specified dimension.

Only the first dimension can be excluded (in declarations of "extern" arrays and function formal parameters).

Error [58]: Identifier 'name' redefined

Redefinition of declarator identifier.

Error [59]: Function returning array

Function cannot return array.

Error [60]: Function definition expected

K&R function header found without following function definition:

```
int f ( i );
```

Error [61]: Missing identifier in declaration

Declarator lacks an identifier.

Error [62]: Simple variable or array of a "void" type

Only pointers, functions and formal parameters can be of "void" type.

Error [63]: Function returning function

Function cannot return function:

```
int f();
```

Error [64]: Unknown size of variable object 'name'

Defined object has unknown size. It could be an external array with no dimension given or an object of an only partially (forward) declared "struct" or "union".

Error [65]: Too many errors encountered (> 100).

Fatal. Compiler aborts after a certain number of diagnostic messages.

Error [66]: Function 'name' redefined

Multiple definitions of function encountered.

Error [67]: Tag 'name' undefined

Definition of variable of "enum" type with type undefined or reference to undefined "struct" or "union" type in function prototype or as "sizeof" argument.

Error [68]: "case" outside "switch"

"case" without any active "switch" statement.

Error [69]: Too many "case" labels (max is 512)

Too many "case" labels in a single "switch" statement.

Error [70]: Duplicated "case" label: nn

The same constant value used more than once as a "case" label.

Error [71]: "default" outside "switch"

"default" without any active "switch" statement.

Error [72]: Multiple "default" within "switch"

More than one "default" in one "switch" statement.

Error [73]: Missing "while" in "do" - "while" statement

Probable cause is missing {} around multiple statements.

Error [74]: Label 'name' redefined

Label defined more than once in the same function.

Error [75]: "continue" outside iteration statement

"continue" outside any active "while", "do - while" or "for" statement.

Error [76]: "break" outside "switch" or iteration statement

"break" outside any active "switch", "while", "do - while" or "for" statement.

Error [77]: Undefined label 'name'

There is "goto label" with no "label:" definition within function's body.

Error [78]: Pointer to a field not allowed

Pointer to a field member of "struct" or "union":

```
struct
{
    int *f:6;
}
```

Error [79]: Argument of binary operator missing

First or second argument of binary operator is missing.

Error [80]: Statement expected

One of '?', ':', ',', ']' or '}' in place where statement is expected.

Error [81]: Declaration after statement

Declaration was found after statement. Note that a single ';' is considered to be an empty statement:

```
int i;    /* Second ';' is a STATEMENT so ... */
char c;   /* this is a declaration after statement */
```

Error [82]: "else" without preceding "if"

Probable cause is bad {} nesting.

Error [83]: "enum" constant(s) outside "int" or "unsigned" "int" range
Too small or too large enumeration constant created.

Error [84]: Too deep "switch" nesting (max is 50)
Fatal. Too many nested "switch" statements.

Error [85]: Empty "struct", "union" or "enum"
Definition of "struct" or "union" that contains no members or definition of "enum" that contains no enumeration constants.

Error [86]: Invalid formal parameter
Declaration of K&R function with formal parameter(s) specified in the function header:

```
int f( c )      /* OK */
char c;
{
    int g( cc ); /* Invalid */
}
```

Error [87]: Redeclared formal parameter: 'name'
A formal parameter in K&R function definition was declared more than once.

Error [88]: Contradictory function declaration
"void" appears in a function parameter type list together with other type of specifiers.

Error [89]: "..." without previous parameter(s)
"..." cannot be the only parameter description specified:

```
int f( ... ); /* Error */

int g( int, ... ); /* OK */
```

Error [90]: Formal parameter identifier missing
No identifier of parameter specified in the header of prototyped function definition:

```
int f( int *p, char, float ff )
    /* Second parameter has no name */
{
    /* Function body */
}
```

Error [91]: Redeclared number of formal parameters

Prototyped function declared with other number of parameters than the first time:

```
int f( int, char );  
int f( int );          /* Less parameters */  
int f( int, char, float ); /* More parameters */
```

Error [92]: Prototype appeared after reference

Prototyped declaration of a function after it was defined or referenced as K&R function.

Error [93]: Initializer to field of width *nn* (bits) out of range

Bit field initialized with constant too large to fit in field's space.

Error [94]: Fields of width 0 must not be named

Zero length fields are only used to align fields to the next "int" boundary and cannot be accessed via an identifier.

Error [95]: Too large difference between "case" values (max is 2000)

Difference between the smallest and the largest "case" value exceeds compiler limit.

Error [96]: "case" label out of range (> 10000 or < -10000)

Value of "case" label exceeds compiler limit.

Error [97]: Undefined "static" function 'name'

Function was declared with "static" storage class but never defined.

Error [98]: Primary expression expected

Missing expression.

Error [100]: Undeclared identifier: 'name'

Reference to identifier other than function that was not declared.

Error [101]: First argument of '.' operator must be of "struct" or "union" type

Dot operator '.' applied to argument that is not "struct" or "union".

Error [102]: First argument of '->' was not pointer to "struct" or "union"

Arrow operator '->' applied to argument that is not pointer to "struct" or "union".

Error [103]: Invalid argument of "sizeof" operator

"sizeof" operator applied to bit field, function or extern array of unknown size.

Error [104]: Initializer "string" exceeds array dimension

Array of "char" with explicit dimension given was initialized with a string exceeding array size:

```
char array [ 4 ] = "abcde";
```

Error [105]: Language feature not implemented: 'name'

The code-generator currently lacks the specified function.

Error [106]: Too many function parameters (max is 127)

Fatal. Too many parameters in function declaration/definition.

Error [107]: Function parameter 'name' already declared

Formal parameter in function definition header declared more than once:

```
int f( i, i )      /* K&R function */
int i;
{
}

int f( int i, int i ) /* Prototyped function */
{
}
```

Error [108]: Function parameter 'name' declared but not found in header

Occurs in K&R function definition. Parameter declared but not specified in the function header:

```
int f( i )
int i, j /* j is not specified in function header */
{
}
```

Error [109]: ';' unexpected

Unexpected delimiter.

Error [110]: ')' unexpected

Unexpected delimiter.

Error [111]: '{' unexpected

Unexpected delimiter.

Error [112]: ',' unexpected

Unexpected delimiter.

Error [113]: ':' unexpected

Unexpected delimiter.

Error [114]: '[' unexpected
Unexpected delimiter.

Error [115]: '(' unexpected
Unexpected delimiter.

Error [116]: Integral expression required
Bad type of expression - evaluated expression must have integral type.

Error [117]: Floating point expression required
Bad type of expression - evaluated expression must have floating type.

Error [118]: Scalar expression required
Bad type of expression - evaluated expression must have scalar type.

Error [119]: Pointer expression required
Bad type of expression - evaluated expression must have pointer type.

Error [120]: Arithmetic expression required
Bad type of expression - evaluated expression must have arithmetic type.

Error [121]: Lvalue required
Expression do not evaluate to a memory address.

Error [122]: Modifiable lvalue required
Expression do not designate a variable object or is "const".

Error [123]: Prototyped function argument number mismatch
Prototyped function called with other number of arguments than declared.

Error [124]: Unknown "struct" or "union" member: 'name'
Reference to nonexistent member of "struct" or "union".

Error [125]: Attempt to take address of field
It is not possible to apply '&' on bit-fields.

Error [126]: Attempt to take address of "register" variable
It is not possible to apply '&' on a object with "register" storage class.

Error [127]: Incompatible pointers

Incompatible pointers. There must be full compatibility of objects that pointers point to. It means that if pointers point (directly or indirectly) to prototyped functions, a compatibility test is performed not only on return values but includes as well compatibility test on number of parameters and their types so sometimes incompatibility can be hidden quite deeply:

```
char>(*p1)[8])(int);
char(*p2)[8])(float);
/* p1 incompatible with p2; incompatible function parameters */
```

Compatibility test includes also checking of dimensions of arrays if they appear in description of objects pointed to:

```
int(*p1)[8]; /* p1 incompatible with p2; */
int(*p2)[9]; /* array dimensions differ */
```

Error [128]: Function argument incompatible with its prototype

Function argument does not match declaration.

Error [129]: Incompatible operands of binary operator

Type conflict in binary operator.

Error [130]: Incompatible operands of '=' operator

Type conflict in assignment.

Error [131]: Incompatible "return" expression

Expression is incompatible with the "return" value declaration.

Error [132]: Incompatible initializer

Initializer expression is incompatible with the object to be initialized.

Error [133]: Constant value required

Expression was not constant in "case" label, #if, #elif, bit-field declarator, array declarator or static initializer.

Error [134]: Unmatching "struct" or "union" arguments to '?' operator

The second and third argument of the '?'-operator are different.

Error [135]: "pointer + pointer" operation

It is an error to add two pointers.

Error [136]: Redclaration error

Current declaration is inconsistent with earlier declarations of the same object.

Error [137]: Reference to member of undefined "struct" or "union"

Only pointers may be declared pointing to undefined "struct" or "union" declarators.

Error [138]: "- pointer" expression

Pointer expression preceded by '-' in other context than "pointer - pointer"

Error [139]: Too many "extern" symbols declared (max is 32768).

Fatal. Compiler limit exceeded.

Error [140]: "void" pointer not allowed in this context

A pointer expression like indexing involved a void pointer (element size unknown).

Error [141]: #error 'any message'

Fatal. The pre-processor directive #error was found which notifies that something must be defined at command-line in order to compile this module.

Error [142]: "interrupt" functions can only be void and have no parameters.

Fatal. A function declared using the "interrupt" keyword is not void or has parameters.

Error [143]: Too large, negative, or overlapping "interrupt [value]" in 'NAME'

The interrupt vector table entry 'value' is too large, negative, or overlaps another vector table entry.

Error [144]: Bad context for storage modifier (storage class or function)

The no_init keyword can only be used to declare variables with static storage-class. That is, no_init cannot be used in typedefs or applied to auto variables of functions. An active #pragma memory=no_init can cause such an error when function declarations are found.

Error [145]: Bad context for function call modifier

The keywords interrupt, non_banked, or monitor can only be applied to function declarations.

Error [146]: Unknown #pragma identifier:'name'

Error [147]: Extension keyword "name" is already defined by the user.

This error will occur if a keyword that can serve as an extension keyword is used as an ordinary identifier (when the compiler is executing in the ANSI mode) and the directive `#pragma language = extended` is found.

Error [148]: '=' expected**Error [149]: Attempt to take address of "sfr" or "bit" address****Error [150]: Illegal range for "sfr" or "bit" address****Error [151]: Too many functions defined in a single module (max is 256)**

The maximum number of function definitions in a single module is 256. Note, however, that there is no limit on the number of function declarations.

Error [152]: '.' expected

Bad bit declaration syntax

Error [153]: Illegal context for "bit" and "sfr" specifier**Error [154]: Macro 'name' redefined**

B.2 C Warning Messages

Warning [0]: Macro 'name' redefined

A `#defined` symbol was redeclared with different argument or formal list.

Warning [1]: Macro formal parameter 'name' is never referenced

A `#define` formal parameter never appeared in the argument string.

Warning [2]: Macro 'name' is already `#undef`

An `#undef` is performed on a non-macro symbol.

Warning [3]: Macro 'name' called with empty parameter(s)

A parameterized `#defined` macro was called with a zero-length argument.

Warning [4]: Macro 'name' is called recursively; not expanded

A recursive macro makes pre-processor stop further expansion.

Warning [5]: Undefined symbol 'name' in #if or #elif; assumed zero

It is considered as bad programming practice to assume that non-macro symbols should be treated as zeros in #if and #elif expressions. Use either:

`#ifdef symbol` or `#if defined (symbol)`

Warning [6]: Unknown escape sequence ("\c"); assumed 'c'

A backslash (\) found in a character constant or string literal was followed by an unknown escape character.

Warning [7]: Nested comment found without using the '-C' option

The character sequence `/*` was found within a comment. Ignored.

Warning [8]: Invalid type-specifier for field; assumed "int"

Bit-fields may in this implementation only be specified as "int" or "unsigned" "int".

Warning [9]: Undeclared function parameter 'name'; assumed "int"

An undeclared identifier in the header of a K&R function definition is by default set to "int".

Warning [10]: Dimension of array ignored; array assumed pointer

An array with an explicit dimension was specified as a formal parameter. It was rewritten to read: pointer to object.

Warning [11]: Storage class "static" ignored; 'name' declared "extern"

An object or function was first declared as "extern" (explicitly or by default) and later declared as "static" which is ignored.

Warning [12]: Incompletely bracketed initializer

Initializers should either be 'flat' (only one level of `{}`) or completely surrounded by curly brackets in order to avoid ambiguity.

Warning [13]: Unreferenced label 'name'

Label was defined but never referenced.

Warning [14]: Type specifier missing; assumed "int"

No type specifier given in declaration - assumed to be "int".

Warning [15]: Wrong usage of string operator ('#' or '##'); ignored

The current implementation restricts usage of '#' and '##' operators to the token-field of parameterized macros. In addition the '#' operator must precede a formal parameter:

```
#define mac(p1)      #p1      /* Becomes "p1" */  
#define mac(p1,p2)  p1+p2##add_this  
/* Merged p2 */
```

Warning [16]: Non-void function: "return" with <expression>; expected

A non-void function definition is supposed to exit with a defined return value in all places.

Warning [17]: Invalid storage class for function; assumed to be "extern"

Invalid storage class for function - ignored. Valid is "extern", "static" or "typedef".

Warning [18]: Redeclared parameter's storage class

Storage class of a function formal parameter changed from "register" to "auto" or vice versa in a subsequent declaration/definition.

Warning [19]: Storage class "extern" ignored; 'name' was first declared as "static"

An identifier declared as "static" was later explicitly or implicitly declared as "extern".

Warning [20]: Unreachable statement(s)

Statement(s) was preceded by an unconditional jump or return:

```
break;  
i = 2;          /* Never executed */
```

Warning [21]: Unreachable statement(s) at unreferenced label 'name'

Labeled statement(s) was preceded by an unconditional jump or return but the label was never referenced:

```
break;  
here:  
i = 2;          /* Never executed */
```

Warning [22]: Non-void function: explicit "return" <expression>; expected

A non-void function generated an implicit return. Could be the result of an unexpected exit from a loop or switch. Note that a "switch" without "default" is always supposed to be 'excitable' regardless of any "case" constructs.

Warning [23]: Undeclared function 'name'; assumed "extern" "int"

Reference to undeclared function causes default declaration. Function is assumed to be of K&R type, having "extern" storage class and returning "int".

Warning [24]: Static memory option converts local "auto" or "register" to "static"

A command line option for static memory allocation was activated which makes "auto" and "register" declarations read as "static".

Warning [25]: Inconsistent use of K&R function - varying number of parameters

K&R function called with changing number of parameters.

Warning [26]: Inconsistent use of K&R function - changing type of parameter

K&R function called with changing types of parameters:

```
f( 34 );      /* Integral argument */  
f( 34.6 );   /* Floating argument */
```

Warning [27]: Size of "extern" object 'name' is unknown

The global type-check option requires that "extern" arrays are declared with size.

Warning [28]: Constant [index] outside array bounds

Constant index outside declared array bounds (< 0 or >= dim).

Warning [29]: Aggregate cannot be "register"

Objects of array, structure or union type cannot have "register" storage class.

Warning [30]: Attribute ignored

Since "const" or "volatile" are attributes of objects they are ignored when given with structure, union or enumeration tag definition with no objects declared at the same time:

```
const struct s
{
    ...
}; /* No object declared; "const" ignored */
```

Warning [31]: Incompatible parameters of K&R functions

Pointers (could be indirect) to functions or K&R function declarators directly used in one of following contexts:

```
"pointer - pointer",
"expression ? ptr : ptr",
"pointer relational_op pointer"
"pointer equality_op pointer",
"pointer = pointer" or
"formal parameter vs actual parameter"
```

have incompatible parameter types.

Warning [32]: Incompatible numbers of parameters of K&R functions

Pointers (could be indirect) to functions or K&R function declarators directly used in one of following contexts:

```
"pointer - pointer",
"expression ? ptr : ptr",
"pointer relational_op pointer"
"pointer equality_op pointer",
"pointer = pointer" or
"formal parameter vs actual parameter"
```

have different number of parameters.

Warning [33]: Local or formal 'name' was never referenced

Formal parameter or local variable object is unused in function definition.

Warning [34]: Non-printable character '\xhh' found in literal or character constant

It is considered as bad programming practice to use non-printable characters in string literals or character constants. Use \0xhh to get the same function.

Warning [35]: Old-style (K&R) type of function declarator

During compilation with the -gA option on, obsolete function declarators was found.

Warning [36]: Floating point constant out of range.

Warning [37]: Illegal floating point operation: division by zero not allowed.

Warning [38]: Tag identifier 'name' was never defined
Structure or union was forward declared but never defined.

APPENDIX - C

Assembler Error Messages

Below is a list, showing the possible error messages you can get while using the A6801 assembler.

Error: Internal align error

Should never occur, if no other error messages have been issued.

Error: Invalid character

Unrecognized token.

Error: Too long line

There must be no more than 132 characters in a line. This applies also to macro generated lines.

Error: Invalid digit

Bad numeric constant.

Error: Space or end-of-line expected

Bad delimiter after the last operand or instruction.

Error: Value out of range

Address or data expression too large or too small.

Error: Syntax error

Undecodable statement.

Error: Expression too complex

Too deep stack needed to evaluate the expression. Re-arrange or simplify expression.

Error: Invalid label

See section "1.4 Label syntax".

Error: Duplicate label

Double definition.

Error: Undefined label: xxxxx

No definition.

Error: Invalid string constant

Bad characters in string constant. See general section "Integer, ASCII and real constants".

Error: Invalid instruction

No such instruction.

Error: Invalid operand combination

Valid operands, but not for this instruction.

Error: Missing END-statement

There must be an END somewhere.

Error: Type conflict

Something about the type (relocation mode) is wrong.

Error: Unmatched NAME, MODULE, END or ENDMOD

Like NAME followed by NAME, instead of END or ENDMOD.

Error: Unmatched MACRO or ENDMAC

Valid sequence is MACRO - ENDMAC.

Error: Bad ELSE or ENDIF nesting

See general section "Conditional assembly".

Error: Over or underflow in macro stack, v=nnn

Macro stack pointer is less than zero or greater than 100.

Error: Macro parameter out of range

Trying to access parameters outside P0-P9.

Error: Unknown "\ " directive in macro

See general section "Macro processing".

Error: Multiple or invalid declaration

See general sections "Modules" and "Segments".

Error: Too deep file nesting, > 3

The limit on include files.

Error: No such file

Could not open include file. Stops assembler.

Error: Expression < > current relocation

Operand must belong to current segment.

Error: Expression is not absolute

Operand must be an absolute expression.

Error: Operator not allowed here

Bad use of SFx or LOW, HIGH, LWRD or HWRD operators. See general section "Expression and operators".

Error: Too many externals or segments defined

256 is the limit.

APPENDIX - D

Linker Error and Warning Messages

The error messages produced by the linker fall into five categories:

- Linker warning messages
- Linker error messages
- Linker fatal error messages
- Memory overflow message
- Linker internal errors

Linker Warning Messages

Linker warning messages will appear when XLINK detects something possibly wrong. The code generated may still be correct.

Linker Error Messages

Linker error messages are produced when XLINK detects something wrong. The linking process will not be aborted but the code produced is probably faulty.

Linker Fatal Errors

Linker fatal error messages abort the linking process. They occur when continued linking is useless, i.e. the fault is irrecoverable.

Memory Overflow Message

The Micro-Series XLINK is a memory-based linker. If run on a system with a small main memory or if very large source files are being used, XLINK may run out of memory. This is recognized by the special message:

```
***LINKER OUT OF MEMORY***
```

Dynamic memory used: nnnnnn bytes

If this occurs, the cure is either to add system memory or to enable file bound processing (see -m and -t options in 10.3 XLINK Command Detail).

Also see section 9.6.2 Host Memory Management for additional actions that can be taken to conserve host system memory.

Linker Internal Errors

During linking, a number of internal consistency checks are performed. If any of these checks fail, the linker will terminate after giving a short description of the problem. These errors will normally not occur but if they do, please report this to Archimedes Software. Please include all possible information about the problem and, also a diskette with the program that generated the error.

D.1 Linker Error Messages

If you get a message that indicates a corrupt object file, reassemble or recompile the faulty file since an interrupted assembly or compilation may produce an invalid object file

Error [0]: Format chosen cannot support banking

Fatal. Format unable to support banking. See discussion of bank-switching under the -b command in 10.3 XLINK Command Detail.

Error [1]: Corrupt file. Unexpected end of file in module *module (file)* encountered

Fatal. Linker aborts immediately.

Error [2]: Too many errors encountered (>100)

Fatal. Linker aborts immediately.

Error [3]: Corrupt file. Checksum failed in module *module* (*file*).
Linker checksum is *linkcheck*, module checksum is *modcheck*
Fatal. Linker aborts immediately.

Error [4]: Corrupt file. Zero length identifier encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [5]: Address type for CPU incorrect. Error encountered in module *module* (*file*).
Fatal. Linker aborts immediately.

Error [6]: Program module *module* declared twice, redeclaration in file *file*. Ignoring second module
Not fatal. XLINK will not produce code, unless option -B (forced dump) is used.

Error [7]: Corrupt file. Unexpected UBROF-format end of file encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [8]: Corrupt file. Unknown or misplaced tag encountered in module *module* (*file*). Tag *tag*.
Fatal. Linker aborts immediately.

Error [9]: Corrupt file. Module *module* start unexpected in file *file*.
Fatal. Linker aborts immediately.

Error [10]: Corrupt file. Segment no. *sno no* declared twice in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [11]: Corrupt file. External no. *ext no* declared twice in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [12]: Unable to open file *file*
Fatal. Linker aborts immediately.

Error [13]: Corrupt file. Error tag encountered in module *module* (*file*)
Fatal. A UBROF error tag was encountered. Linker aborts immediately.

Error [14]: Corrupt file. Local *local* defined twice in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [15]: Faulty bank definition -*bbank* def
 Fatal. Incorrect syntax. Linker aborts immediately.

Error [16]: Segment *segment* is too long for segment definition
 Fatal. The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.

Error [17]: Segment *segment* is defined twice in segment definition -*Zsegdef*
 Fatal. Linker aborts immediately.

Error [18]: Range error in module *module* (*file*), segment *segment* at address *address*. Value *value*, in tag *tag*, is out of bounds
 Not fatal. The address is out of the CPU address range. An example of how this can happen is:

```
address code
(A relative segment)
0000  nop           If this segment is defined, or
0001  nop           allocated, to start at address F700
. .               then the address to LABEL1 will be
. .               F700 + 0FC7 = 106C7. If the CPU
0025  nop           address range is 16-bits (range 0-FFFF)
0026  JUMP LABEL1  it will be unable to reach
0027  nop           LABEL1. The segment will therefore
. .               partly be out of the CPU address
. .               range.
0FC6  LABEL1:
0FC7  nop
. .
```

This check can be suppressed by use of the switch -R (disable range check).

Note that you should never get this message on a well-designed program. Always try to find the exact position or instruction that caused the problem by using the information in the message (file:module:segment:address).

Error [19]: Corrupt file. Undefined segment referenced in module *module* (*file*)
 Fatal. Linker aborts immediately.

Error [20]: Undefined external referenced in module *module* (*file*)
 Fatal. Linker aborts immediately.

Error [21]: Segment *segment* in module *module* does not fit bank
 Fatal. The segment is too long. Linker aborts immediately.

Error [22]: Paragraph no. is not applicable for the wanted CPU.
Tag encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [23]: Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module *module* (*file*)
Fatal. The tag T_REL_FI_8 or T_EXT_FI_8 is faulty. Linker aborts immediately.

Error [24]: Segment *segment* overlaps segment *segment*
The segments overlap each other. i.e both have code on the same address.

Error [25]: Corrupt file. Unable to find module *module* (*file*)
Fatal. A module is missing. Linker aborts immediately.

Error [26]: Segment *segment* is too long
Fatal. This error should never occur, unless the program is extremely large. Linker aborts immediately.

Error [27]: Entry *entry* in module *module* (*file*) redefined in module *module* (*file*)
Fatal. There are two or more entries with the same name. Linker aborts immediately.

Error [28]: File *file* is too long
Fatal. This error should never occur unless the program is extremely large. Linker aborts immediately.

Error [29]: No object file specified in command line
Fatal. There is nothing to link. Linker aborts immediately.

Error [30]: Option *-option* also requires the *-option* option
Fatal. Linker aborts immediately.

Error [31]: Option *-option* cannot be combined with the *-option* option
Fatal. Linker aborts immediately.

Error [32]: Option *-option* cannot be combined with the *-option* option and the *-option* option
Fatal. Linker aborts immediately.

Error [33]: Faulty value *val* (in command line or in XLINK PAGE), (range is 10-150)
Fatal. Faulty page setting. Linker aborts immediately.

Error [34]: Filename too long

Fatal. The filename is more than 255 characters long. Linker aborts immediately.

Error [35]: Unknown flag *flag* in cross reference option *option*

Fatal. Linker aborts immediately.

Error [36]: Option *op* does not exist

Fatal. Linker aborts immediately.

Error [37]: - not succeeded by character

Fatal. The '-' marks the beginning of an option, and must be followed by a character. Linker aborts immediately.

Error [38]: Option *option* multiply defined

Fatal. Linker aborts immediately.

Error [39]: Illegal character specified in option *op*

Fatal. Linker aborts immediately.

Error [40]: Argument expected after option *op*

Fatal. This option must be succeeded by an argument. Linker aborts immediately.

Error [41]: Unexpected '-' in option *op*

Fatal. Linker aborts immediately.

Error [42]: Faulty symbol definition -*Dsymbol* definition

Fatal. Incorrect syntax. Linker aborts immediately.

Error [43]: Symbol in symbol definition too long

Fatal. The symbol name is more than 255 characters. Linker aborts immediately.

Error [44]: Faulty value *val* (in command line or in XLINK COLUMNS), (range 80-300)

Fatal. Faulty column setting. Linker aborts immediately.

Error [45]: Unknown CPU *CPU* encountered in command line (or in XLINK CPU)

Fatal. Linker aborts immediately.

Error [46]: Undefined external *external* referred in module *module* (*file*)

Not fatal. Entry to external is missing.

If you get this message with the external name ?CLxxx_3_yy_Lzz when using an Archimedes C-compiler, you are either using the

wrong (probably an old) version of the library, or using the wrong library for the particular processor or memory model option selected (for the module name specified in this error message)

Error [47]: Unknown format *format* encountered in command line or XLINK FORMAT

Fatal. Linker aborts immediately.

Error [48]: Faulty segment definition *-Zsegdef*

Fatal. Incorrect syntax. Linker aborts immediately.

Error [49]: Segment *name* in segment definition too long

Fatal. The segment name is more than 255 characters long. Linker aborts immediately.

Error [50]: Paragraph no. not allowed for this CPU, encountered in option *option*

Fatal. Linker aborts immediately.

Error [51]: Hexadecimal or decimal value expected in option *option*

Fatal. Linker aborts immediately.

Error [52]: Overflow on value in option *option*

Fatal. Linker aborts immediately.

Error [53]: Parameter exceeded 255 characters in extended command line file *file*

Fatal. Linker aborts immediately.

Error [54]: Extended command line file *file* is empty

Fatal. Linker aborts immediately.

Error [55]: Extended command line variable XLINK_ENVPAR is empty

Fatal. Linker aborts immediately.

Error [56]: Overlapping ranges in segment definition *segment def*

Fatal. Linker aborts immediately.

Error [57]: No CPU defined

Fatal. No CPU defined, neither in command line nor in XLINK_CPU. Linker aborts immediately.

Error [58]: No format defined

Fatal. No format defined, neither in command line nor in XLINK_FORMAT. Linker aborts immediately.

Error [59]: Revision no. for *file* is incompatible with XLINK revision no.

Fatal. Linker aborts immediately.

If this error exists after recompilation or reassembly the wrong version of XLINK is used (check with distributor)

Error [60]: Segment *segment* defined in bank definition and segment definition.

Fatal. Linker aborts immediately.

Error [61]: Symbol in bank definition is too long

Fatal. Linker aborts immediately.

Error [62]: File *file* multiply defined in command line

Fatal. Linker aborts immediately.

Error [63]: Trying to pop an empty stack in module *module* (*file*)

Fatal. Linker aborts immediately.

Error [64]: module *module* (*file*) has not the same debug type as the other modules

Fatal. Linker aborts immediately.

Error [65]: Faulty replacement definition *-replacement* definition

Fatal. Incorrect syntax. Linker aborts immediately.

Error [79]: Faulty allocation definition *-adefinition*

Syntax error in the *-a* modifiers

Error [80]: Symbol in allocation definition (*-a*) too long

Syntax error in the *-a* modifiers

Error [81]: Unknown flag in cross reference option *-Y/4/2*

Error [82]: Conflict in segment '*name*'. Mixing overlayable and non-overlayable segment parts.

Error [83]: The overlayable segment '*name*' may not be banked

Error [84]: The overlayable segment '*name*' must of relative type

D.2 Linker Warning Messages

Warning [0]: Too many warnings
Too many warnings encountered.

Warning [1]: Error tag encountered in module *module* (*file*)
An UBROF error tag was encountered when loading file *file*. This indicates a corrupt file and will generate an error in the linking phase.

Warning [2]: Symbol *symbol* is redefined in command line
The linker warns on redefinition of symbols.

Warning [3]: Type conflict. Segment *segment*, in module *module*, is incompatible with earlier segment(s) of the same name.
Segment of the same name should have the same type.

Warning [4]: Close/open conflict. Segment *segment*, in module *module*, is incompatible with earlier segment of the same name.
Segments of the same name should be either open or closed.

Warning [5]: Segment *segment* cannot be combined with previous segment
The segments will not be combined.

Warning [6]: Type conflict for external/entry *entry*, in module *module*, against external/entry in module *module*
Entries and their corresponding externals should have the same type.

Note that you may get this message on C programs that calls C library routines without using the proper `#include` header file

```
main()
{
    printf("Don't forget to #include stdio.h!");
}
```

Also note that the `-c` compiler option can generate such warnings in conjunction with C library function calls.

Warning [7]: Module *module* declared twice, once as program and once as library. Redefined in file *file*, ignoring library module
The program module is linked.

Warning [8]: Segment *segment* undefined in segment or bank definition
Undefined segment exists. All segments should be defined in either the segment or the bank definition

Warning [9]: Ignoring redeclared program entry
Only the first found program entry is chosen.

Warning [10]: No modules to link
The linker has no modules to link.

Warning [11]: Module *module* declared twice as library. Redeclared in file *file*, ignoring second module
The first found module is linked.

Warning [12]: Using SFB in banked segment *segment* in module *module* (*file*)
The SFB assembler directive may not work in a banked segment.

Warning [13]: Using SFE in banked segment *segment* in module *module* (*file*)
The SFE assembler directive may not work in a banked segment.

Warning [14]: Entry *entry* duplicated. Module *module* (*file*) loaded, module *module* (*file*) discarded
Duplicated entries exist and concerned modules are conditionally loaded, i.e. library modules or conditionally loaded program modules (option -C).

APPENDIX - E

Librarian Error Messages

Error messages are of two kinds, command errors and fatal errors. Command errors only abort the current command, while fatal errors make XLIB abort. In addition to the messages below, Pascal (the implementation language) I/O errors may also occur. These messages are normally easy to interpret. Commands flagged as erroneous never alter object files!

Error: Bad object file, EOF encountered

Fatal. Bad or empty object file, could be the result of an aborted assembly.

Error: Unexpected EOF in batch file

Fatal. The last command in a command file must be EXIT.

Error: File open error

Fatal. Could not open the command file, or if ON-ERROR-EXIT has been specified, this message is issued on any failing file open operation.

Error: Variable length record out of bounds

Fatal. Bad object module, could be the result of an aborted assembly.

Error: Missing or non-default parameter

A parameter was missing in the "direct mode". Look into section 4.11

Error: No such CPU

A list with the possible choices is listed when this error is found. 6801 family CPU must be specified.

Error: CPU undefined

DEFINE-CPU must be issued before object file operations can begin. A list with the possible choices is listed when this error is found. 6801 family CPU must be specified.

Error: Ambiguous CPU type

A list with the possible choices is listed when this error is found. 6801 family CPU must be specified.

Error: No such command

Use the HELP command or look into section 4.11

Error: Ambiguous command

Use the HELP command or look into section 4.11

Error: Invalid parameter(s)

Too many parameters or a misspelled parameter.

Error: Module out of sequence

Fatal. Bad object module, could be the result of an aborted assembly.

Error: Incompatible object.

Fatal. Bad object module, could be the result of an aborted assembly, or that the used assembler/compiler revision, is incompatible with the version of XLIB used.

Error: Unknown tag: hh

Fatal. Bad object module, could be the result of an aborted assembly.

Error: Too many errors

Fatal. More than 32 errors will make XLIB abort.

Error: Assembly/compilation error?

Fatal. The T_ERROR tag was found. Edit and re-assemble/compile your program.

Error: Bad CRC, hhhh expected

Fatal. Bad object module, could be the result of an aborted assembly.

Error: No help file!

Fatal. Could not find the file with the help text. Look into section 4.13 for the fix.

Error: Can't find module: xxxxx

Check what's available with LIST-MOD THEFILE.

Error: Module expression out of range

Module expression is less than one or greater than <\$>.

Error: Bad syntax in module expression: xxxxx

Look into section 4.6

Error: No SCRATCH!

Fatal. Could not create a temporary file. Possible cause: Directory is full.

Error: INTERNAL ERROR IN PROCEDURE: xxx/nnn,
Fatal. If no other fatal errors have occurred so far, this should never happen.

Error: <End module> found before <Start module>!
Source module range must be from low to high order.

Error: Before or after!
Bad <Before/After> specifier in the INSERT-MODULES command.

Error: Illegal insert sequence
The specified <Dest mod> in the INSERT-MODULES command, must not be within the <Start mod> - <End mod> sequence.

APPENDIX - F

Libraries

The Archimedes 68HC11 C-compiler kit includes most of the important C-library functions that apply to microcontrollers, i.e. PROM-based embedded systems. The small and large models use the cl6811.r07 file and the banked model uses the cl6811b.r07 file.

Before using any of these library functions you must include the appropriate header for that function in your C program. The header file (e.g., #include <ctype.h>) must be declared before any reference to the functions or objects it declares, or to types or macros it defines.

The floating point library contains the most popular advanced math functions. Review the header math.h.

The C-compiler also has special C-run time libraries (?xxxx functions).

The following library functions are available:

VARIABLE ARGUMENTS <stdarg.h>

va_arg, va_end, va_start

LOW-LEVEL ROUTINES <icclbutl.h>

_formatted_write, _medium_write, _small_write, formatted_read, _medium_read

COMMON DEFINITIONS <stddef.h>

Various usable definitions like size_t, NULL, ptrdiff_t, offsetof etc.

INTEGRAL TYPES <limits.h>

Limits/sizes on integral types

FLOATING POINT TYPES <float.h>

Limits/sizes on floating point types

ERRORS <errno.h>

Error return values

INTERRUPTS <int6811.h>

Definitions of interrupt vector addresses for the 68HC11

CHARACTER HANDLING <ctype.h>

isalnum, isalpha, iscntrl, isdigit, isgraph, isxdigit, islower, isprint, ispunct, isspace, isupper, tolower, toupper

NON-LOGICAL JUMPS <setjmp.h>

longjmp, setjmp

FORMATTED INPUT/OUTPUT <stdio.h>

getchar, gets, scanf, sscanf, printf, putchar, sprintf,

GENERAL UTILITIES <stdlib.h>

atof, atoi, atol, exit, calloc, free, malloc, realloc

STRING HANDLING <string.h>

strcat, strcmp, strcpy, strlen, strncat, strncmp, strncpy

MATHEMATICS <math.h>

atan, atan2, cos, exp, log, log10, modf, pow, sin, sqrt, tan

Source code is provided for all the formatted input/output library functions to allow modification for a particular target environment. All library functions above are summarized in alphabetical order below. For a detailed description you may also reference any standard C language reference book.

Summary of Library Functions

abort()

Purpose:

To terminate the program abnormally.

Prototype:

```
#include <stdlib.h>
void abort(void);
```

abs()

Purpose:

The abs function computes the absolute value of an integer.

Prototype:

```
#include <math.h>
int abs(int j);
```

acos()

Purpose:

The acos function computes the principal value of the arc cosine of x. Argument should be in the range [-1, +1]. The return value is in the range [0, pi].

Prototype:

```
#include <math.h>
double acos(double x);
```

asin()

Purpose:

The asin function computes the principal value of the arc sine of x . The argument should be in the range $[-1, +1]$. The return value is in the range $[-\pi/2, +\pi/2]$.

Prototype:

```
#include <math.h>
double asin(double x);
```

atan()

Purpose:

Computes the arc tangent of x . Returns a value in the range $[-\pi/2, \pi/2]$.

Prototype:

```
#include <math.h>
double atan(double x)
```

atan2

Purpose:

Computes the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value. Returns a value in the range $[-\pi, \pi]$.

Prototype:

```
#include <math.h>
double atan2(double y, double x);
```

atof**Purpose:**

Converts a number given by the ASCII string `nptr` into a double. The `atof` function skips white-space and terminates reading input as soon as something unknown is found. That is, " -3K", ".0006" and "1e-4 " would generate -3.00, 0.0006 and 0.0001 respectively.

Prototype:

```
#include <stdlib.h>
double atof(const char *nptr)
```

atoi**Purpose:**

Converts a decimal number given by the ASCII string `nptr` into an int. The `atoi` function skips white-space and terminates reading input as soon as something unknown is found. That is, " -3K", "6" and "149 " would generate -3, 6 and 149 respectively.

Prototype:

```
#include <stdlib.h>
int atoi(const char *nptr);
```

atol**Purpose:**

Converts a decimal number given by the ASCII string `nptr` into a long. The `atol` function skips white-space and terminates reading input as soon as something unknown is found. That is, " -3K", "6" and "149 " would generate -3, 6 and 149 respectively.

Prototype:

```
#include <stdlib.h>
long atol(const char *nptr)
```

calloc**Purpose:**

Allocates space for an array of nmemb objects. Size of elements in bytes is specified by size. For details concerning changing the default heap size see the target dependent section discussing the heap. Calloc returns a pointer (or zero if failed to find any suitable memory) to the start (lowest byte address) of the object.

Prototype:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

ceil()**Purpose:**

Computes and returns the smallest integral value not less than x.

Prototype:

```
#include <math.h>
double ceil(double x);
```

cos()**Purpose:**

Computes and returns the cosine of x (measured in radians).

Prototype:

```
#include <math.h>
double cos(double x);
```

div()**Purpose:**

Computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The `div` function returns a structure of type `div_t`, comprising both the quotient and the remainder (see `stdlib.h` for the structure). `quot * denom + rem` shall equal `numer`.

Prototype:

```
#include <math.h>
div_t div(int numer, int denom);
```

exit()**Purpose:**

Normal program termination. No return to its caller.

Prototype:

```
#include <stdlib.h>
void exit(int status);
```

exp()**Purpose:**

Computes and returns the exponential function of x.

Prototype:

```
#include <math.h>
double exp(double x);
```

fabs()**Purpose:**

computes the absolute value of a floating-point number.

Prototype:

```
#include <math.h>
double fabs(double x);
```

floor()**Purpose:**

Computes and returns the largest integral value not greater than x.

Prototype:

```
#include <math.h>
double floor(double x);
```

fmod()**Purpose:**

Computes the floating-point remainder of x/y , i.e. the value $x - i * y$, for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y .

Prototype:

```
#include <math.h>
double fmod(double x, double y);
```

_formatted_read**Prototype:**

```
#include <icclbutl.h>
int _formatted_read (const char **line, const char **format,
va_list ap);
```

Purpose:

The `scanf` and `sscanf` routines share a common formatter called `_formatted_read`. Due to the code size and run time stack size required by this full ANSI version of the formatter, a smaller version of the formatter is installed in the C libraries. The two versions are:

_formatted_read

Supports full ANSI `scanf/sscanf` definition.

_medium_read

Differs from ANSI version by not supporting floating point numbers. It is about half the size of the full formatter. If the specifiers `%f`, `%g`, `%G`, `%e` or `%E` are used when the `_medium_read` formatter is chosen the linker will respond with the message : `FLOATS? wrong formatter installed!`

Installation of the smaller version of the formatter is specified in the linker control files (see example lnk*.xcl files) by including the command line :

`-e _medium_read = _formatted_read`

If no line is specified the full ANSI formatter will be the default selection.

`_formatted_write`

Prototype:

```
#include <icclbutl.h>
int _formatted_write(const char *format, void outputf(char, void
*),void *sp,va_list ap);
```

The printf and sprintf routines share a common formatter called formatted write. Due to the code size and run time stack size required by this full ANSI version of the formatter, three different formatters are actually installed in the C library. These are :

`_formatted_write`

Supports full ANSI printf/sprintf definition.

`_medium_write`

Differs from ANSI version by not supporting floating point numbers. It is about half the size of the full formatter. If the specifiers %f, %g, %G, %e or %E are used when the `_medium_write` formatter is chosen the linker will respond with the message :FLOATS? wrong formatter installed!

`_small_write`

A small version of the formatter that only supports the basic %, %d, %o, %c, %s and %x specifiers for int type objects and does not support field width or precision arguments. The size of this routine is 20% of `_formatted_write`.

Which formatter to use is specified in the linker control files (see examples in the supplied lnk*.xcl files) by including a command line such as :

```
-e _small_write= _formatted_write
```

If no line is specified the full ANSI formatter will be the default selection.

As an alternative to the built-in printf/sprintf formatters there is a C source file intwri.c included in the current release diskettes. The file is an integer only version of the formatter and can be modified by the user to support particular formatting needs or non-standard output devices. Note that this formatter includes the entire printf function and therefore does not directly support sprintf.

free()

Purpose:

Object pointed to by ptr is made available for further allocation. ptr must earlier have been assigned a value from malloc, calloc, or realloc. Returns no value.

Prototype:

```
#include <stdlib.h>
void free(void *ptr);
```

getchar()

Purpose:

(Source is included). Gets a character from standard input. Returns next character from input stream. Delivered in source format for easy adaption to the target hardware configuration.

Prototype:

```
#include <stdio.h>
int getchar(void);
```

gets()**Purpose:**

Gets an end-of-line terminated string from standard input and puts it into *s. Returns a null pointer on EOF. The end-of-line character is replaced by '\0' and a pointer is returned with the same value as the input buffer on successful calls to gets. Calls getchar.

Prototype:

```
#include <stdio.h>
char *gets(char *s);
```

isalnum()**Purpose:**

Tests for any letter or digit. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isalnum(int c);
```

isalpha()**Purpose:**

Tests for any letter. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isalpha(int c);
```

isctrl()**Purpose:**

Tests for any control character. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isctrl(int c);
```

isdigit()**Purpose:**

Tests for any decimal digit. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isdigit(int c);
```

isgraph()**Purpose:**

Tests for printable character not including space (' '). Returns non-zero if the test is true.

Prototype:

```
#include <ctype.h>
int isgraph(int c);
```

islower()**Purpose:**

Tests for any lower-case letter. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int islower(int c);
```

isprint()**Purpose:**

Tests for any printable character (including space). Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isprint(int c);
```

ispunct()**Purpose:**

Tests for any printable character except space, a digit or a letter. Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int ispunct(int c);
```

isspace()**Purpose:**

Tests for the following characters: ' ' (space), '\f' (form feed), '\n' (new line), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab). Returns non-zero if test is true.

Prototype:

```
#include <ctype.h>
int isspace (int c);
```

isupper()**Purpose:**

Tests for any upper-case letters, returns non-zero if true.

Prototype:

```
#include <ctype.h>
int isupper(int c);
```

isxdigit()**Purpose:**

Tests for hexadecimal digit (0-9, a-f, A-F). Returns non-zero if true.

Prototype:

```
#include <ctype.h>
int isxdigit(int c);
```

labs()**Purpose:**

computes the absolute value of an long integer j.

Prototype:

```
#include <stdlib.h>
long int labs(long int j);
```

ldexp()**Purpose:**

The ldexp function returns the value of the floating-point number x multiplied by 2 raised by an integral exp.

Prototype:

```
#include <math.h>
double ldexp(double x,int exp);
```

ldiv()**Purpose:**

Computes the quotient and remainder of the division of the numerator numer by the denominator denom. If the division is inexact, the resulting quotient is the long integer of lesser magnitude that is the nearest to the algebraic quotient. The div function returns a structure of type ldiv_t, comprising

both the quotient and the remainder (see `stdlib.h` for the structure).

$\text{quot} * \text{denom} + \text{rem} = \text{numer}$.

Prototype:

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

log()

Purpose:

Computes and returns the natural logarithm of x .

Prototype:

```
#include <math.h>
double log(double x);
```

log10()

Purpose:

Computes and returns the base-ten logarithm of x .

Prototype:

```
#include <math.h>
double log10(double x);
```

longjmp()

Purpose:

Restores what was saved by the last call to setjmp (corresponding jmp_buf argument).

Prototype:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

malloc()

Purpose:

Allocates space for an object. Its size in bytes is specified by size. For details concerning changing the default heap size see the target dependent section discussing the heap. Malloc returns a pointer (or zero if failed to find any suitable memory) to the start (lowest byte address) of the object.

Prototype:

```
#include <stdlib.h>
void *malloc(size_t size);
```

memchr()

Purpose:

The memchr locates the first occurrence of c (converted to an unsigned char) in the initial n characters (each interpreted as unsigned char) of the object pointed to by s. It returns a pointer to the located character, or a null pointer if the character does not occur in the object.

Prototype:

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

memcmp()**Purpose:**

The memcmp function compares the first n characters of the object pointed to by s1 to the first n characters of the object pointed to by s2. It returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

Prototype:

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

memcpy()**Purpose:**

The memcpy function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined (see memmove). It returns the value of s1.

Prototype:

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

memmove()

Purpose:

The memmove function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1* (see memcopy). It returns the value of *s1*.

Prototype:

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

memset()

Purpose:

The memset function copies the value of *c* (converted to an unsigned char) into each of the first *n* characters of the object pointed to by *s*. It returns the value of *s*.

Prototype:

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

modf()

Purpose:

Computes the integral (stored in *iptr) and the fractional parts of value (function return). Sign of both parts is the same as for value.

Prototype:

```
#include <math.h>
double modf(double value, double *iptr);
```

pow()**Purpose:**

Computes and returns x raised to the power of y.

Prototype:

```
#include <math.h>
double pow(double x, double y);
```

printf()**Prototype:**

```
#include <stdio.h>
int printf(const char *format, ...);
```

A summary of the NEW printf conversion specifiers providing full ANSI support are shown below. Each conversion specification is introduced by %. Conversion specifications within brackets are optional.

% {flags} {field_width} {.precision} {length_modifier} conversion

flags

- # Alternate form. Has the following effect:
 - For o (octal) specifiers the first digit will always be a zero.
 - G, g, E, e and f specifiers will always print a decimal point.
 - G and g specifiers will also keep trailing zeros.
 - X and x (hex) will pre-append non-zero values with 0x (or 0X).
- 0 Zero padding to field width (for d, i, o, u, x, X, e, E, f, g, and G specifiers).

field width

Number of characters to be printed in the field. Field width will be padded with space if needed. A negative value indicates a left-adjusted field. If given as '*' the next argument should be an integer holding the field width.

.precision

Number of digits to print for integers (d, i, o, u, x and X). Number of decimals printed for floating point values (e, E, and f) and number of significant digits for g and G conversions. If given as '*' the next argument should be an integer holding the precision.

length_modifier

- h Used before d, i, u, x, X, or o specifiers to denote a short, int, or unsigned short int value
- L Used before e, E, f, g or G specifiers to denote a long double value.

conversion

- g Write floating point constant in f or e notation depending on the size of the value ("best" fit specifier).
- G Write floating point constant in f or E notation depending on the size of the value ("best" fit specifier).
- n Store current number of characters written so far. Argument should be a pointer to integer.

Note that promotion rules convert all char and short int arguments to int while floats are converted to double.

Examples:

Assume the following declarations were compiled:

```
int i=5, j=-6;
char *p = "ABC";
long l=100000;
float f1 = 0.0000001, f2 = 750000;
double d = 2.2;
```


Then execute the following calls to printf:

printf-statement	printed line	number of chars
printf("%c",p[1]);	B	1
printf("%d",i);	5	1
printf("%3d",i);	5	3
printf("%.3d",i);	005	3
printf("%-10.3d",i);	005	10
printf("%10.3d",i);	005	10
printf("Value=%+3d",i);	Value= +5	9
printf("%10.*d",i,j);	-00006	10
printf("String=[%s]",p);	String=[ABC]	12
printf("Value=%lX",l);	186A0	5
printf("%f",f1);	0.000000	8
printf("%f",f2);	750000.000000	13
printf("%e",f1);	1.000000e-07	12
printf("%20e",d);	2.200000e+00	20
printf("%.4e",d);	2.2000e+00	10
printf("%g",f1);	1e-07	5
printf("%g",f2);	750000	6
printf("%g",d);	2.2	3

A reduced version of printf is delivered in source code intended for customization (see file intwri.c).

putchar()

Purpose:

(Source is included). Writes the character specified by value to standard output. May be implemented as a macro. Returns EOF (-1) if errors occur, otherwise the character written. This routine is delivered in source format for adaption to the target hardware configuration. Note that putchar also serves as the low-level output function in printf.

Prototype:

```
#include <stdio.h>
int putchar(int value);
```

puts()

Purpose:

The puts function writes the string pointed to by *s* to the standard output, and appends a new-line character to the output. It returns EOF (-1) if errors occur, otherwise it returns a nonnegative value.

Prototype:

```
#include <stdio.h>
int puts(const char *s);
```

rand()

Purpose:

The rand function computes a sequence of pseudo-random integers in the 0 to RAND_MAX. It returns a pseudo-random integer. See srand for a description of the pseudo-random sequences.

Prototype:

```
#include <stdlib.h>
int rand(void);
```

realloc()

Purpose:

Changes the size of the object pointed to by *ptr* (which must have got its value from malloc, calloc, or realloc) to the size specified by *size*. Realloc returns a pointer (may be zero if no room in heap left) to the start address of the possibly moved object.

Prototype:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

scanf()**Prototype:**

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Purpose:

Reads formatted data from standard input and optionally assigns converted data to variables specified by the format string.

Returns the number of successful conversions (or EOF if input is exhausted).

Variables must always be expressed as addresses in order to be assignable by scanf.

Note that input is performed through library function gets which in turn calls getchar which must be adapted for the actual target hardware configuration.

If the format string contains white-space characters, input is scanned until a non-white-space character is found.

A conversion specification is introduced by %.

If the format string neither contains a white-space or a %, the format string and the input character must match exactly.

A summary of the scanf conversion specifiers is shown below. Conversion specifications within brackets are optional.

% {assign_suppress} {field_width} {length_modifier} conversion

assign_suppress

* NO assignment should be done (just scan the field)

field width

Maximum field to be scanned (default is until no match occurs).

length_modifier

- l** (letter ell) Used before d, i, or n to indicate long int (default is int) or before o, u, or x to denote the presence of an unsigned long int.
For e, E, g, G, and f conversions the l character implies a double operand (default is float).
- h** Used before d, i, or n to indicate short int (default is int) or before o, u, or x to denote the presence of an unsigned short int.
- L** For e, E, g, G, and f conversions the L character implies a long double operand (default is float).

conversion

- d** Read an optionally signed decimal integer value
- i** Read an optionally signed integer value in standard C notation. That is, default is decimal notation but octal (0n) and hexadecimal (0xn, 0Xn) notations are also recognized
- o** Read an optionally signed octal integer
- u** Read an unsigned decimal integer
- x** Read an optionally signed hexadecimal integer
- X** Equivalent to x
- f** Read floating point constant
- e** Equivalent to f
- E** Equivalent to f
- g** Equivalent to f
- G** Equivalent to f
- s** Read character string
- c** Read one or field_width characters
- n** Store number of characters read so far Argument should be a pointer to integer
- p** Read pointer value (address)
- [** Read characters as long as they match any of the characters that are within the terminating]. If the first character after [is a ^ the matching condition is reversed. If the [is immediately followed by] or ^] the] is assumed to belong to the matching sequence (i.e. there must be another terminating] character)
- %** Read a % character

Except for the `l`, `c` or `n` specifiers leading white-space characters are skipped.

Examples:

Assume the following declarations were compiled:

```
int n, i;  
char name[50];  
float x;
```

Then the following call to `scanf`:

```
n = scanf("%d%f%s", &i, &x, name);
```

with an input line

```
25 54.32E-1 Hello World
```

will set `n = 3`, `i = 25`, `x = 5.432` and `name=Hello World\0` while

```
scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with an input line

```
56789 0123 56a72
```

will set `i = 56`, `x = 789.0` and `name=56\0`.

Note that `%*d` just scanned 0123 without assigning.

srand()**Purpose:**

The `srand` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand`. If then `srand` is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If `rand` is called before any calls to `srand` have been made, the same sequence shall be generated as when `srand` is first called with a seed value of 1.

Prototype:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

sscanf()**Purpose:**

Equivalent to scanf except that the argument s specifies a string from which the input is taken instead of from an I/O-stream.

Prototype:

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

va_start(),va_arg(),va_end()**Purpose:**

Macros for taking parameters in a portable fashion from variable parameter lists (specified with ...).

Prototype:

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

setjmp()

Purpose:

Saves its calling environment in its jmp_buf argument for later use by longjmp. Returns the value zero if not returned from a longjmp.

Prototype:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

sin()

Purpose:

Computes and returns the sine of x (measured in radians).

Prototype:

```
#include <math.h>
double sin(double x);
```

sprintf()

Purpose:

(Source is included). Writes formatted data to a character array as specified by s. For details concerning formatting see printf.

Prototype:

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

strcat()**Purpose:**

Appends a copy of the string pointed to by s2 to the end of the string pointed to by s1. Returns value of s1.

Prototype:

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

strchr()**Purpose:**

The strchr function locates the first occurrence of c (converted to a char) in the string pointed to by s. The terminating null character is considered to be part of the string. It returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Prototype:

```
#include <string.h>
char *strchr(const char *s, int c);
```

strcmp()**Purpose:**

Compares the string pointed to by s1 to the string pointed to by s2. Returns an integer value greater than, equal to or less than zero if the string pointed to by s1 is greater than equal to, or less than string pointed to by s2.

Prototype:

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

strcoll()**Purpose:**

The strcoll function compares the string pointed to by s1 to the string pointed to by s2. It returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

Prototype:

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

strcpy()**Purpose:**

Copies the string pointed to by s2 into the object pointed to by s1. Returns the value of s1.

Prototype:

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

strcspn()**Purpose:**

The strcspn function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters not from the string pointed to by s2. It returns the length.

Prototype:

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

strlen()**Purpose:**

Calculates the number of characters in the string pointed to by *s* (not including NULL character). Returns the number of initial characters.

Prototype:

```
#include <string.h>
size_t strlen(const char *s);
```

sqrt()**Purpose:**

Computes and returns the square root of *x*.

Prototype:

```
#include <math.h>
double sqrt(double x);
```

strncat()**Purpose:**

Appends a copy of the string (up to *n* characters) pointed to by *s2* to the end of the string pointed to by *s1*. Returns the value of *s1*.

Prototype:

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

strncmp()

Purpose:

Compares the string (up to *n* characters) pointed to by *s1* to the string pointed to by *s2*. Returns an integer value greater than, equal to or less than zero if the string pointed to by *s1* is greater than equal to, or less than string pointed to by *s2*.

Prototype:

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

strncpy()

Purpose:

Copies the string (up to *n* characters) pointed to by *s2* into the object pointed to by *s1*. Returns the value of *s1*.

Prototype:

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

strpbrk()

Purpose:

The *strpbrk* function locates the first occurrence in the string pointed to by *s1* of any character from the string pointed to by *s2*. It returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

Prototype:

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

strrchr()

Purpose:

The `strrchr` function locates the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string. It returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Prototype:

```
#include <string.h>
char *strrchr(const char *s, int c);
```

strspn()

Purpose:

The `strspn` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`. It returns the length.

Prototype:

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

strstr()

Purpose:

The `strstr` function locates the first occurrence in the string pointed to by `s1` of the sequence of characters (excluding the terminating null character) in the string pointed to by `s2`. It returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length, the function returns `s1`.

Prototype:

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

strtod()**Purpose:**

The strtod function converts a sequence, resembling a floating-point constant, found in the string pointed to by *nptr* to a double. The function returns the double, or if no floating-point constant is found it returns zero. The *endptr* will point to the first character in the string after the floating-point constant if there is such a constant, else *endptr* will have the value of *nptr*.

Prototype:

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

strtol()**Purpose:**

The strtol function converts a sequence, resembling an integer constant, found in the string pointed to by *nptr* to a long integer. The function returns the long integer, or if no integer constant is found it returns zero. The *endptr* will point to the first character in the string after the integer constant if there is such a constant, else *endptr* will have the value of *nptr*.

If the base is zero the sequence expected is an ordinary integer, else the expected sequence is an integer consisting of digits and letters representing an integer with the radix specified by base (must be between 2 and 36). The letters from a (A) through z (Z) are ascribed the values 10 through 35. If the base is 16, the 0x portion of an hex integer is allowed as initial sequence.

Prototype:

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

strtoul()**Purpose:**

The strtoul function converts a sequence, resembling an integer constant, found in the string pointed to by nptr to an unsigned long integer. The function returns the unsigned long integer, or if no integer constant is found it returns zero. The endptr will point to the first character in the string after the integer constant if there is such a constant, else endptr will have the value of nptr.

If the base is zero the sequence expected is an ordinary integer, else the expected sequence is an integer consisting of digits and letters representing an integer with the radix specified by base (must be between 2 and 36). The letters from a (A) through z (Z) are ascribed the values 10 through 35. If the base is 16, the 0x portion of an hex integer is allowed as initial sequence.

Prototype:

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, bases
int);
```

tan()**Purpose:**

Computes and returns the tangent of x (measured in radians).

Prototype:

```
#include <math.h>
double tan(double x);
```

tolower()**Purpose:**

Converts an upper-case letter to the corresponding lower-case letter.

Prototype:

```
#include <ctype.h>
int tolower(int c);
```

toupper()**Purpose:**

Converts a lower-case letter to the corresponding upper-case letter.

Prototype:

```
#include <ctype.h>
int toupper(int c);
```


APPENDIX - G

Proliferation Chip Support

The Archimedes 68HC11 C software kit supports development of any chip based on the 68HC11 basic architecture (i.e. any future chip that uses the 68HC11 instruction set).

Pre-defined register block addresses of the core 68HC11 are in header file io6811.h. As an example, I/O port A is defined as:

```
# define PORT A (*(char *) (HC_REG_OFFSET + 0))
```

Review io6811.h for more detailed information. You may add new features of future 68HC11 proliferation chips in this header file.

The assembler's complete instruction set is listed in the file a6801ins.s07.

APPENDIX - H

Special Emulator Support

Refer to the EMULATOR.DOC file on disk #1 of the distribution diskettes for more updated information.

APPENDIX - I

Code Example

This Appendix shows a code example detailing the usage of some Archimedes C 68HC11 constructs. Please review the tutorial section for additional details. The complete code example is available on the distribution media.

I.1 EXAMPLE.BAT

```
c-6811 example -ml -g -L -q -e -rn -F -p53
a6801 asfunc asfunc
xlink -f example
```

I.2 EXAMPLE.C

```
/* EXAMPLE.C: Sample code for Archimedes 68HC11 compiler kit.
   Can be run on Motorola 68HC11 EVB board */

#include <stdio.h>          /* header for printf and putchar */
#include <io6811.h>         /* header with internal address definitions */

#define TRUE 1
#define FALSE 0
#define EXT_PORT (*(char *) 0xA000) /* defines external port
                                       at location A000 hex */
#define EEPROM (char *)0xB600      /* ptr to EEPROM base */

extern int asfunc (int parm1);      /* declaration for sample assembl
                                       routine -- see ASMFUNC.S07 */

void delay(void)                  /* delay about 12 ms */
{
    int i;
    for (i = 0; i < 400; i++);
}

void eeprom_write(char data, char *address)
{
    PPROG = 0x16;                 /* single-byte erase mode */
    *address = 0xff; /* write anything */
    PPROG = 0x17;

    delay();
}
```

```

        PPROG = 0x16;

        PPROG = 0x02;          /* program mode */
        *address = data; /* write data */
        PPROG = 0x03;
        delay();
        PPROG = 0x02;
        PPROG = 0;              /* read mode */
    }

void dump() /* Displays misc. data for illustration.
            * Note: Specific internal memory locations and
            * registers and absolute external memory locations
            * are accessed using C pointers as shown below.
            * See the I06811.H file for predefined register
            * names for the 68HC11 (PORT1, PORT2 etc.).
            * Also, see section 1.14 in the manual.
            */
{
    int line, byte;

    /* dump internal RAM */
    printf("6811 internal RAM dump:\n");
    for(byte = 0; byte < 16; printf("%X ", byte++));
    for(line = 0; line < 16; line++)
    {
        printf("\n%02X: ", line * 16);
        for(byte = 0; byte < 16; byte++)
            printf("%02X ", *(char *)((line * 16) + byte));
    }
    /* read PORT A, PORT E and an external port */
    printf("\nPORT A = %02X \nPORT E = %02X\n", PORTA, PORTE);
    printf("Port or memory at address A000h = %02Xh\n", EXT_PORT);
    printf("First byte in EEROM = %02X\n", *EEPROM);
}

#define SIZE 2048
char flags[SIZE+1]; /* array for sieve */

void sieve() /* Eratosthenese Sieve program from BYTE, 1/83 */
{
    register int i, k; /* register class NOT supported -- ignored */
    int prime, count, iter;

    printf("\nSieve: 10 iterations...\n");
    for (iter = 1; iter <= 10; iter++) /* do program 10 times */
    {
        count = 0; /* initialize prime counter */
        for (i = 0; i <= SIZE; i++) /* set all flags true */
            flags[i] = TRUE;
        for (i = 0; i <= SIZE; i++)
        {
            if (flags[i]) /* found a prime */
            {
                prime = i + i + 3; /* twice index + 3 */
                for (k = i + prime; k <= SIZE; k += prime)

```

```

        flags[k] = FALSE;      /* kill all multiples */
        count++;      /* primes found */
    }
}

printf("%d primes, sieve done.\n",count); /* found in 10th pass */
}

void main() /* main example.c program */
{
    int parm1;

    disable_interrupt();
    printf("EXAMPLE.C sample program for 68HC11.\n");
    while(1)
    {
        eeprom_write(0xA5,EEPROM); /* write A5h to first byte of EEPROM
        dump(); /* display memory, ports */
        parm1 = 10; /* set up parameter */
        asmfunc (parm1); /* call sample assembler routine */
        sieve(); /* do the Sieve */
    }
}

```

I.3 ASMFUNC.S07

```

*-----*
*  asmfunc.s07 -- dummy assembler function  *
*                                           *
*  int asmfunc(int parm1);                *
*                                           *
*  This is a dummy routine that demonstrates how *
*  to interface C with an assembler function.  *
*  Called from example.c program.          *
*                                           *
*-----*

```

```

MODULE    asmfunc
PUBLIC    asmfunc

```

```

P68H11    Use for 68HC11 chip
RSEG      CODE

```

```

asmfunc:
*  Insert your assembler routine here.
*  The first parameter (parm1 in this case) is now
*  in the A:B register pair (B is the low byte).
*  This first parameter, and any following parameters
*  that were declared, are also on the stack and may be
*  pulled off after saving the return address, which is
*  on the top of the stack. (see section 1.9 in the manual).
*
*  A return value, if needed, should be placed in the B or
*  A:B registers.

```

RTS
END

Return to C program

I.4 EXAMPLE.XCL

```
-!                               -LNK6811.XCL-

XLINK 4.xx command file to be used with the 68HC11 C-compiler V3.xx
using the -ms or -ml options (non banked memory models).
Usage: xlink your_file(s) -f lnk6811

First define CPU -!

-c68hc11

-! Allocate segments which should be loaded -!

-! First allocate the read only segments.
C000 was here supposed to be start of PROM -!

-Z(CODE)RCODE, CODE, CDATA, ZVECT, CONST, CSTR, CCSTR=C000

-! Then the writeable segments which must be mapped to a RAM area
2000 was here supposed to be start of RAM.
Note: Stack size is set to 128 (80H) bytes with 'CSTACK+80'
!

-Z(DATA)DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP, CSTACK+80

-! The interrupt vectors are assumed to start at FFD6, see also
CSTARTUP.S07 and INT6811.H. -!

-Z(DATA)INTVEC=FFD6

-! NOTE: In case of a RAM-only system, the two segment lists may be
connected to allocate a contiguous memory space. I.e. :
-Z...CCSTR, DATA...=start_of_RAM -!

-! The segment SHORTAD is for direct addressing, let XLINK check
that the variables really are within the zero page -!

-Z(DATA)SHORTAD=00-FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write

-! See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read

-! Now load the 'C' library -!
cl6811 -! or cl6811d -!
example
```


asmfunc

```
-! Code will now reside in example.a07 in MOTOROLA 'S' format -!
-o example.a07
-! Create a map file with a full cross reference list -!
-x
-l example.map
```

I.5 CSTARTUP.S07

```
*-----*
*
*                                CSTARTUP.S07
*
* This module contains the 6301/68HC11 startup routine and
* must usually be tailored to suit special hardware needs
*
* Note: The routine ?SEG_INIT_L07 is now included in CSTARTUP
* The size of stack is set in the link-file
* The segment INTVEC is declared COMMON
*
* Version: 3.30 [IAHW 11/Feb/92]
*-----*
```

```
NAME      CSTARTUP
$defcpu.inc
EXTERN    ?C_EXIT    Where to go when program is done
EXTERN    main       Where to begin execution
```

```
*-----*
* CSTACK - The C stack segment
*
* Please look in the link-file lnk6???.xcl how to increment
* the stack without having to reassemble cstartup.s07 !
*-----*
```

```
RSEG      CSTACK
RMB       0          A bare minimum !!
```

```
*-----*
* Forward declarations of segments used in initialization
*-----*
```

```
RSEG      UDATA
RSEG      IDATA
RSEG      ECSTR
RSEG      TEMP
RSEG      DATA
RSEG      WCSTR
RSEG      CDATA
RSEG      ZVECT
RSEG      CCSTR
RSEG      CONST
RSEG      CSTR
```

```

*-----*
*  RCODE - Where the program actually starts  *
*-----*

      RSEG    RCODE
init_C:
      LDS     #.SFE.(CSTACK) - 1           From high to low addresses

*-----*
* If the 68HC11 OPTION register MUST be modified, here is the *
* place to do it in order to meet the 64-cycle requirement.   *
*-----*

*-----*
* If it is not a requirement that static/global data is set  *
* to zero or to some explicit value at startup, the next line *
* of code can be deleted.                                     *
*-----*

      BSR     seg_init

*-----*
* If hardware must be initiated from assembly or if interrupts *
* should be on when reaching main, this is the place to insert *
* such code.                                                    *
*-----*

      IF      banking

      EXTERN  ?X_CALL_L09
      LDX     #main
      JSR     ?X_CALL_L09          main()

      ELSE

      JSR     main                main()

      ENDIF

*-----*
* Now when we are ready with our C program (usually 6301/6811 *
* programs are continuous) we must perform a system-dependent *
* action. In this simple case we jump to ?C_EXIT.              *
*-----*
* DO NOT CHANGE NEXT LINE OF CSTARTUP IF YOU WANT TO RUN YOUR *
* SOFTWARE WITH THE AID OF THE C-SPY HLL DEBUGGER.             *
*-----*

      JMP     ?C_EXIT

```

```

*-----*
* Copy initialized PROMmed code to shadow RAM and clear
* uninitialized variables.
*-----*

seg_init:

*-----*
* Zero out UDATA
*-----*

        LDX    #.SFE.(UDATA)
        LDD    #.SFB.(UDATA)
        BSR    zero_mem

*-----*
* Zero out ZVECT's
*-----*

        LDX    #.SFB.(ZVECT)
        PSHX
        LDX    #.SFE.(ZVECT)
        PSHX
        TSX

more_ZVECT:
        LDD    0,X                load end address
        SUBD   2,X
        BEQ    no_more_ZVECT
        LDX    2,X
        LDD    0,X                D - start zero field
        LDX    2,X                X = End zero field + 1
        BSR    zero_mem
        TSX                        increase start address
        LDD    #4
        ADDD   2,X
        STD    2,X
        BRA    more_ZVECT

no_more_ZVECT:

*-----*
* Copy CDATA into IDATA
*-----*

        LDD    #.SFE.(CDATA)
        STD    2,X
        LDD    #.SFB.(CDATA)
        STD    0,X
        LDX    #.SFB.(IDATA)
        PSHX
        BSR    copy_mem

*-----*
* Copy CCSTR into ECSTR
*-----*

```

```

TSX
LDD    #.SFE.(CCSTR)
STD    4,X
LDD    #.SFB.(CCSTR)
STD    2,X
LDD    #.SFB.(ECSTR)
STD    0,X
BSR    copy_mem
INS
INS
INS
INS
INS
INS
RTS                                End of initialization

*-----*
* Clear memory                      *
*-----*

zero_mem:
_PSHB
_PSHA
_PSHX
TSX
again:
LDD    0,X
SUBD   2,X
BEQ    return
LDX    2,X
CLR    0,X
INX
%XGDX
TSX
STD    2,X
BRA    again
return:
INS
INS
INS
INS
RTS

*-----*
* Copy memory                      *
*-----*

copy_mem:
TSX
more_copy:
LDD    6,X
SUBD   4,X
BEQ    quit_copy
LDX    4,X
LDAA   0,X
_PSHA

```

```

        INX
        %XGDY
        TSX
        STD      5,X
        PULA
        LDY      3,X
        STAA     0,X
        INX
        %XGDY
        TSX
        STD      2,X
        BRA      more_copy
quit_copy:
        RTS

*-----*
* Interrupt vectors must be inserted by the user.   Here we *
* only used RESET.                                     *
*-----*

        COMMON  INTVEC
        IF      proc6811
*
        RMB     40          Assuming start address = FFD6 for 68HC11

        ELSE
*
        RMB     20          Assuming start address = FFEA for 6301

        ENDIF

        FDB     init_C

        ENDMOD   init_C      'init_C' is program entry address

*-----*
* Function/module: exit(int code)                      *
*                                                         *
* When C-SPY is used this code will automatically be replaced *
* by a 'debug' version of exit().                        *
*-----*

        MODULE   exit

        PUBLIC   exit
        PUBLIC   ?C_EXIT

        RSEG     RCODE

?C_EXIT:

```

```

*-----*
* The next line could be replaced by user defined code.      *
*-----*

```

```

end_loop:
    BRA    end_loop

```

```

    IF     banking

```

```

    RSEG   FLIST

```

```

exit:
    FQB    ?C_EXIT

```

```

    ELSE

```

```

    exitEQU ?C_EXIT

```

```

    ENDIF

```

```

    END

```

APPENDIX - J

Glossary of Terms

The terms included here are those that are specific to, or especially important in, C-6811. Refer to a C textbook for general C language definitions.

ANSI-C

The "official" definition for the C language proposed by the American National Standards Institute, number X3J11.

Auto

A C variable is said to be an auto variable if it is declared within the body of a function. The memory space of an auto variable is dynamically allocated on the stack when a function is called, and de-allocated when the function exits. Auto variables are only supported in the reentrant memory models; they are converted to "static" by the compiler if a static model is used.

Cross-software

Software development tools that run on one type of computer or processor and produce code for another, different, type of processor. For example, C-6811 cross-compiler and cross-assembler run on an 8088/86-based PC/AT or VAX-based host processor and produces code for the Motorola 68HC11 microcontroller.

Global

A variable is said to be global if it is declared outside any function body. Global variable are "static", i.e., they reside in a fixed location in memory. Only global variables (and functions) are normally included in the symbol table output by the linker.

Helper functions

These are C-6811 library routines that implement various low-level runtime functions that cannot be efficiently implemented with compiler-generated inline code (e.g. long division). Calls to these routines are transparently and automatically generated by the compiler. Note that all helper routine names are prefaced with a "?" (e.g., ?L_DVMD_L15).

Host

Refers to the computer/processor used as the development system to run the C-6811 compiler, assembler, linker, etc.

In-line functions

Those functions that generate in-line code rather than a subroutine call. The C-6811 functions, such as enable_interrupt, disable_interrupt, and wait_for_interrupt, are in-line functions. These functions are enabled with the -e compiler switch.

K&R

Refers to the "old" definition of C as contained in the book "The C programming language", by Brian Kernighan and Dennis Ritchie of Bell Labs.

Keyword

Any word that has a special meaning to the C compiler and, therefore, cannot be used as a variable, function or constant name.

Library

A C or Assembly module may have a "library" or "program" attribute. A library module is a module that will be loaded by the linker only if it contains at least one entry that is referenced by another module that has already been loaded. The compiler switch -b gives a C module the library attribute, while the MODULE directive gives a library attribute to an assembly program. All modules contained in

the library files CL6811?.R07 (except for CSTARTUP) have a library attribute.

Macro

A macro is a single C or assembly statement that the macro pre-processor (a part of the compiler or the assembler) expands into a series of in-line statements that perform an often used task. C macros are created with the `#define` statement, while `MACRO` is used to create assembly macros.

Module

A function or group of functions that makes up part or all of a C-6811 program. With the C-6811 compiler, a single module is produced by each C source file that is compiled. All functions that belong to that file share the same module name. However, a single assembly file can contain more than one module. A module is made up of one or more function and data element.

Program

A C or assembly module may have a "program" or "library" attribute. A program module is a module that will always be loaded by the linker when the file to which it belongs is processed. By default, a C module has a "program" attribute. The `NAME` directive is used to give assembly modules a "program" attribute. The only module in the library files CL6811?.R07 that has the program attribute is CSTARTUP.

Reentrant

A function is said to be reentrant if it can be interrupted and called again by the interrupt service routine. The reentrant memory models must be used whenever reentrancy is needed. Reentrant functions use the stack for local variable allocations.

Recursive

A function is said to be recursive if it calls itself directly or indirectly. The latter case is usually referred to as indirect recursion.

Segment

A segment is a group of code or data elements that are all related in some functional way, so they may be linked together into a program. For example, the segment RCODE contains executable instructions that must reside in a resident (non-banked) read-only memory. C-6811 uses a number of predefined segments listed in the C-Compiler section of this manual. User-defined segments can also be created.

Startup routine (CSTARTUP)

This is the code that the 68HC11 processor executes immediately following a hard reset or a power-on. This code consists of some runtime initialization code essential to the operation of C-6811 programs.

Static

A C variable located in a fixed location in memory that is active throughout the program. This word is also used in the different memory models. Static memory models allocate all local variables statically in memory for more efficiency.

Target

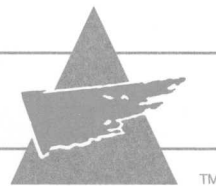
Refers to the processor and its associated hardware that the compiler and assembler generate code for (e.g., Motorola 68HC11).

UBROF

The Universal Binary Relocatable Object Format. This format is an Archimedes-Proprietary format used for output files from the compiler and assembler. The linker, XLINK, reads UBROF files and outputs files in one of more than 20 different formats. Refer to the Linker section of this manual for more detailed information.

The Time Saver

Index



ARCHIMEDES
SOFTWARE

INDEX

- #pragma I-26
 - codeseg I-28
 - function I-28
 - language I-26
 - memory I-26
 - warning I-29
- #pragma keywords I-26
- \$ (Include directive) II-97
- * (Program Location Counter) II-50, II-76
- .DATE. operator II-57
- ! Linker option III-20, III-48
- 2 Compiler option I-39
- _argt\$ I-55
- _argt\$ I-55
- _formatted_read F-9
- _formatted_read I-19
- _formatted_write F-10
- _formatted_write I-18
- _medium_read I-19
- _medium_write I-18
- _opc() I-31
- _small_write I-18
- _TID_ I-55
- ?ABS_ENTRY_MOD III-55
- = (Assembler directive) II-83
- ? (Assembler option) II-13
- 6811 T-1
- 68HC11
 - CPU type in Linker III-11, III-54
 - Overlay checking in Linker III-13
 - CPU type in Librarian IV-21, IV-23
 - Architecture II-44
 - Family chips II-2
 - Proliferation chips II-2

A

- a,-A Compiler Option I-43
 - see compiler, options
- A Linker Option III-49 add index
- a Linker option III-50 add index
- A6801 T-8
 - See Assembler
- A6801.EXE II-9
- Abort() F-3
- Abs() F-3
- Absolute code & data II-34, II-76, II-81
- Absolute code & data III-29
- Absolute listings II-22
- Absolute read/write I-10
- Acos() F-3
- ADEMO sample program II-8
- ADEMO1 sample program II-31
- ADEMO2 sample program II-42
- ANSI ADAPTATIONS I-52
- ANSI C I-49
- ANSI-C J-1
- AOMF III-74, III-79
- ASEG (Assembler directive) II-34, II-77
- Asin() F-4
- Assembler T-8
 - Defining segments III-27
 - Basic operation II-6
 - Characteristics II-2
 - Command line examples II-13
 - Command line options II-9, II-11
 - Command line syntax II-9
 - Comment lines II-32
 - Compatibility with other assemblers II-3
 - Constants II-46
 - Development Cycle II-1
 - Directives II-30
 - Directives (Summary) II-65
 - Error messages II-15
 - Expressions II-49
 - Features II-2
 - File naming conventions II-5

- Include files II-45, II-97
- Installation II-4
- Listing controls II-21
- Listings II-7, II-10, II-15, II-20, II-26
- Macros II-25, II-112
- output T-24
- Overview II-1
- Programs
 - Creating II-28
 - Downloading & Testing II-26
 - Editing II-5
 - Linking II-16, III-25
 - Multi-module II-39, II-42, II-84
- Prompted operation II-14
- Relocatable object files II-11
- Sample program II-31
- Source files II-5, II-10, II-28, II-29
- Source lines II-29, II-37
- Assembler directives
 - MODULE T-12
 - NAME T-12
- Assembly language interface I-11
- Assumptions T-1
- Atan() F-4
- Atan2 F-4
- Atof F-5
- Atoi F-5
- Atol F-5
- Attribute
 - Library T-9, T-10, T-12
 - Program T-9, T-10, T-12
- Auto J-1

B

- b Compiler Option I-37
- b, Linker option I-9, III-51, III-53
- Banked memory model I-7
- Banked segments
 - See Segments, Banked
- Batch files III-47, A-6

Battery-powered RAM I-17
 Binary image files III-79, III-84
 Books
 recommended T-2
 BRCLR, usage I-30
 BRSET, usage I-30

C

-c Compiler option I-41
 -C Compiler option I-45
 -c Linker option III-54
 -C Linker option III-53
 C-6811 Compiler
 Bank-switching III-52
 Example .XCL file III-23
 Interfacing with assembler II-1, II-74
 Modules III-26
 Segments III-28
 Type checking III-31
 C_INCLUDE I-46
 Calloc F-6
 Cast T-6
 CCSTR I-16
 CDATA I-16
 Ceil() F-6
 Char, plain I-3
 Chip support T-1
 CL6811*.R07 I-7
 CODE I-16
 Codeseg, #pragma I-28
 Comment lines
 Assembler II-32
 In Librarian command file IV-32
 In Linker command file III-48, III-20
 COMMON (Assembler directive) II-79
 Compiler
 diagnostics I-47
 environment variable
 C_INCLUDE I-46
 QCC_6811 I-45

- errors B-3
 - command line I-47
 - compilation I-47
- extensions I-55
 - _argt\$ I-55
 - _TID_ I-55
- fatal error
 - compilation I-48
- internal error I-48
- memory models I-5
- memory overflow I-48
- operation T-7
- output T-21
- Overview T-1
- operation I-35
- optimization I-2
- options I-36
 - 2 I-39
 - a,-A I-43
 - b I-37
 - c I-3, I-41, I-45
 - d I-39, I-44
 - e I-41
 - F I-42, I-44
 - g I-38, I-45
 - g (type check enable) I-2
 - Hname I-43
 - i I-42, I-43
 - K I-45
 - l,-L I-41
 - m (memory models) I-5
 - o,-O I-37
 - pnn I-42
 - q I-42
 - r I-40, I-43
 - s I-40, I-45
 - s (speed optimizer) I-2
 - T I-42
 - tn I-42
 - U I-44
 - w I-39
 - x I-42

- y I-41
- z I-40
- z (size optimizer) I-2
- overview I-2
- warnings B-16
 - compilation I-47
- Conditional assembly II-93
- CONFIG.SYS file A-2
- Configuration I-18
- CONST I-16, I-49
- Constant pointer, example I-10
- Constants
 - ASCII character II-47
 - In Assembler programs II-38
 - Integer II-46
 - Listing of examples II-90
 - Real number II-47
- Cos() F-7
- CRC II-24
- CRC IV-28
- Cross reference listing
 - See Linker, Listings III-37
- Cross-software J-1
- CSTACK I-17
- CSTARTUP T-9
 - modification I-18
 - replacing in library I-18
- CSTARTUP file III-26
- CSTR I-16

D

- d Compiler option I-39
 - see compiler, options
- D Compiler option I-44
 - see compiler, options
- D Linker option III-55
- d Linker option III-56
- Data (RAM) II-44
- DATA I-17
- Data representation I-3

- double precision I-3
- floating point I-3, I-4
- Hex constants I-3
- single precision I-3
- Data types I-50
 - structures I-52
 - union I-52
- DATE operator IV-28
- Debugging T-3
- DEC VAX host system III-10, III-42
- Def6811.inc I-18
- Def6811b.inc I-18
- Default libraries III-62
- Defcpu.inc I-18
- DEFINE (Assembler directive) II-84
- Define (Assembler listing) II-25
- Define II-108
- Delimiters
 - Command line II-9
 - In Assembler expressions II-51
 - In Assembler source lines II-37
 - In Linker command (.XCL) files III-20
 - In Linker command lines III-9
 - In XLIB commands IV-21
 - Linker command line III-40
- Diagnostics, Compiler I-47
- Directives, Assembler
 - Conditional Assembly II-93
 - Counting II-99
 - Listing control II-98
 - Module definition II-67
 - Segment definition II-75
 - Summary II-65
 - Symbol declaration II-70
 - Value declaration II-82, II-85
- Directives, Compiler pre-processor I-54
- Disable interrupt function I-21
- Div() F-7
- DOS error return codes A-5
- Downloading T-29

E

- e Compiler option I-41
 - see compiler, options
- E Linker option III-57
- e Linker option III-56
- E, Assembler option II-12
- ECSTR I-17
- Editors T-3
- ELSE (Assembler directive) II-95
- Emulators, hardware II-27
- Emulators, hardware III-79
- Enable interrupt function I-21
- END (Assembler directive) II-39, II-69
- ENDIF (Assembler directive) II-95
- ENDMOD (Assembler directive) II-69
- Enum I-50
- Environment variable, compiler I-45
- Environment variables III-15, III-45
- Environment variables IV-20
- EQU (Assembler directive) II-83
- Error messages
 - Assembler II-15
 - Linker III-32, D-1
- Errors, compilation I-47
- Errors, compiler command line I-47
- Example program T-1, T-10
- EXAMPLE.C T-17
- Examples I-31
- Exit() F-7
- Exp() F-8
- Expressions, Assembler
 - Delimiters II-51
 - Involving external symbols II-53
 - Involving relocatable addresses II-52
 - Operands II-50
 - Operators II-51
 - Overview II-49
 - TRUE and FALSE II-52
- Extended language
 - interrupts T-6
- Extensions I-55

EXTERN (Assembler directive) II-36, II-41, II-71

F

-F Compiler option I-42

-f Compiler option I-44

-f Linker option III-58

-F Linker option III-60

F (Assembler option) II-12

Fabs() F-8

FCB (Assembler directive) II-86

FCB (Assembler directive) III-30

FCB, FDB, FCC, FQB (Assembler directive) II-45

FCC (Assembler directive) II-90

Files I-35

 ASMFUNC.S07 T-24

 CL6811*.R07 I-7

 copying T-3

 EXAMPLE.C T-17

 include I-46

 include T-21

 IO68111.H I-10

 standard T-2

 types T-2

 .A07 T-3

 .BAT T-3

 .C T-3

 .DOC T-3

 .EXE T-3

 .H T-3

 .INC T-3

 .LST T-3

 .MAP T-28

 .R07 T-3

 .S07 T-3

 .XCL T-3, T-24

Filetypes II-5, III-5

FLIST segment I-16, I-8

Floating point representation I-3

Floor() F-8

Fmod() F-9

Format, Linker

AOMF8051 III-79
 AOMF8096 III-80
 ASHLING III-80
 DEBUG III-80
 EXTENDED-TECKHEX III-80
 HP-CODE III-82
 HP-SYMBOL III-82
 INTEL-STANDARD III-82
 MILLENIUM III-83
 MOTOROLA III-83
 MODS-CODE III-84
 MPDS-SYMB III-84
 MSD III-85
 NEC-SYMBOLIC III-85
 PENTICA-A,-B,-C,-D III-86
 RCA III-87
 SYMBOLIC, TYPED III-87
 TI7000 (TMS7000) III-89
 UBROF T-7
 ZAX III-89

Formatter I-18, I-19

selecting I-18, I-19

Forward references II-81

FQB (Assembler directive) II-88

Free() F-11

Function

assembly T-23
 prototype T-8, T-18, I-51
 return value I-13

Function, #pragma I-28

G

-g Compiler option I-38
 see compiler, options

-G Compiler option I-45

-G Linker option III-61

GENLAB (in macros) II-118

GETCHAR routine T-15, F-11

Gets() F-12

Global J-1
Global symbols
 See Symbols, Global

H

-Hname Compiler option I-43
Hardware Access T-5
Header files
 IO6811.H I-10
Heap I-19
Helper functions J-2
Host J-2
HP-3000 host system III-10, III-42

I

-i Compiler option I-42
-I Com[iler option I-43
I/O I-10
IDATA I-17
Identifiers I-3
IEEE I-3
IF (Assembler directive) II-94
In-Line assembly I-31
In-line functions I-21, J-2
 disable_interrupt I-21
 enable_interrupt I-21
 wait_for_interrupt I-21
Include files I-46, II-15, II-97
 See Assembler, Include files II-45
Initialization I-18
Installation T-3, A-3
Intel-standard Hex III-82
Interface to Assembly language I-11
Internal error, compiler I-48
Internal errors, Linker III-32
Interrupt T-14, I-20
 examples I-20
 keyword I-20

vector table I-20
 Interrupt, keyword I-23
 INTVEC I-16, I-23
 IO6811.H I-10
 IO6811.H T-6
 Isalnum() F-12
 Isalpha() F-12
 Iscntrl() F-13
 Isdigit() F-13
 Isgraph() F-13
 Islower() F-14
 Isprint() F-14
 Ispunct() F-14
 Isspace() F-15
 Isupper() F-15
 Isxdigit() F-15

K

-K Compiler option I-45
 -K Linker option III-62
 K&R J-2
 see language, K&R
 Keyword J-2
 Keywords
 const I-49
 enum I-50
 signed I-49
 void I-50
 volatile I-49
 Keywords, ANSI C I-49

L

-l, -L Compiler options I-41
 see compiler, options
 -l Linker option III-62
 Labels II-29
 See also, Symbols
 Labs() F-16

- Language, #pragma I-26
- Language, extensions I-53
- Language
 - ANSI C T-4
 - ANSI I-1
 - C I-1
 - C-6811 T-4
 - extensions I-21
 - In-Line assembly I-31
 - interface I-11
 - K&R T-4, I-1
 - keywords
 - Interrupt I-23
 - monitor I-24
 - no_init I-22
 - non_banked I-25
 - zpage I-22
- Ldexp() F-16
- Ldiv() F-16
- Length of identifiers
 - see Identifiers I-3
- Librarian
 - Adding modules IV-13
 - Batch operation IV-15
 - Changing names in a library IV-13
 - Command abbreviations IV-21
 - Command line operation IV-8
 - Command reference IV-23
 - Command summary IV-19
 - Creating a library IV-13
 - Environment variables IV-20
 - File naming conventions IV-7
 - Installation IV-7
 - Interactive operation IV-8
 - Listing library modules IV-9
 - Listing library symbols IV-11
 - Listings IV-17, IV-20
 - Overview of operation IV-1, IV-8
 - overview T-10
 - Replacing library modules IV-12
 - Typical uses for IV-9
- Libraries

- Also see Librarian
- Multi-module assembler II-40
- Library J-2
- Library module
 - See Modules, Library
- Library, attribute T-9, T-10, T-12
- Linker
 - Command (.XCL) file examples III-21
 - Command (.XCL) files III-14, III-19, III-48, III-58
 - Command detail III-48
 - command file I-15
 - command file T-24
 - Command line examples III-16
 - Command line syntax III-6, III-9, III-40
 - Command summary III-43
 - Defining segments (-Z) III-12, III-72
 - DOS return codes III-47
 - Environment variables III-15, III-45
 - Errors and warnings D-1
 - Errors and warnings III-32, III-47, III-68
 - Essential commands III-11
 - Example T-24
 - File naming III-5
 - Input files and modules III-6, III-25, III-37, III-49, III-53
 - Installation III-4
 - Linking assembler programs II-16
 - Linking library modules III-26
 - Listings III-13, III-34, III-62, III-66, III-69
 - Memory management III-33, III-63, III-68
 - Numeric formats III-42
 - options III-48
 - output T-9
 - .MAP T-9
 - Output files III-13, III-53, III-65
 - Output formats III-14, III-60, III-74, III-77
 - overview T-9
 - Overview of operation III-1
 - segments I-16, I-17
 - segments, other names I-17
 - Segment location III-27
 - Specifications and Features III-2
- Linker Options III-48

Linking I-14
LINT T-8, I-3
Listings
 See Assembler or Linker, Listings
LNK6811.XCL I-15
Local symbols
 See Symbols, Local
LOCSYM (Assembler directive) II-75
Log() F-17
Log10() F-17
Longjmp() F-18
LSTCND (Assembler directive) II-100
LSTCOD (Assembler directive) II-101
LSTEXP (Assembler directive) II-102
LSTFOR (Assembler directive) II-105
LSTMAC (Assembler directive) II-101
LSTOUT (Assembler directive) II-99
LSTPAG (Assembler directive) II-106
LSTWID (Assembler directive) II-103
LSTXRF (Assembler directive) II-108

M

-mb Compiler option
 see compiler, options
-ml Compiler option
 see compiler, options
-ms Compiler option
 see compiler, options
-m Linker option III-63
MACRO (Assembler directive) II-113
Macro J-3
Macros, Assembler
 Advanced features II-117
 Defining II-113
 Examples of II-126
 Macro basics II-112
 Operators (\ commands) II-119
 Parameters II-115
Malloc I-19, F-18
Memchr() F-18

- Memcmp() F-19
- Memcpy() F-19
- Memmove() F-20
- Memory models I-5
 - banked T-4, I-7
 - large T-4
 - reentrant T-5, I-5
 - small T-4
 - static T-5, I-5
- Memory requirements A-1
- Memory use (host) III-33, III-63
- Memory, #pragma I-26
- Memset() F-20
- Modf() F-20
- MODULE (Assembler directive) T-12, II-33, II-68
- Module J-3
 - attribute
 - Library T-9
 - Program T-9
 - CSTARTUP T-9, T-11
 - replace T-15
- Modules
 - Changing type from Prog to Lib IV-31
 - Display list of in a file IV-30
 - In Assembler listings II-24
 - In Assembler programs II-29, II-32, II-41
 - In Linker listing map III-37
 - Library type II-40, II-68
 - Library type III-26, III-37, III-53
 - Library type IV-1, IV-4
 - Linking III-25
 - Listing in a library IV-9
 - Naming II-32, II-68
 - Naming III-37
 - Overview II-67
 - Program type II-32, II-40, II-68
 - Program type III-26, III-37, III-53
 - Program type IV-1
 - Renaming in Librarian IV-34
- Monitor functions I-24
- Monitor, keyword I-24
- Motorola S-Record Format II-8

MSTACK (in macros) II-118

N

-n Linker option III-64

NAME Assembler directive T-12, II-32, II-68

NAME T-11

see also Assembler,directives

NO_INIT I-17, I-22, I-23

Non_banked functions I-25

Non_banked, keyword I-25

Non_volatile memory I-22

O

-o, -O Compiler Options I-37

-o Linker option III-65

see Linker, options

Object files, relocatable

See Relocatable object files

Operators, Assembler

Detailed reference II-55

Precedence II-53

Summary table II-54

Options, Assembler

See Assembler, Command line options

ORG Assembler Directive II-76, II-81

Out of memory, compiler I-48

Output formats

See Linker, Output formats

Overlay checking III-13, III-71

Overlay, segments III-50

P

-P Compiler option I-38

see compiler, options

-pnn Compiler option I-42

-p Linker option III-66

P Assembler option II-11
PAGSIZ Assembler directive II-106
Parameter Stack I-12
Pointers Vs integers I-50
Pow() F-21
Pre-processor compiler directives I-54
Printf I-18, F-21
Program entry point II-39, II-69, III-37
Program J-3
Program location counter II-50, II-76
Program module
 See Module, Program type
Program, attribute T-9, T-10, T-12
PROM programmers II-8, II-27, III-58, III-79
PROMable Code T-5
Pseudo-ops
 See Assembler, Directives II-30
PSTITL Assembler directive II-108
PTITL Assembler directive II-107
PUBLIC Assembler directive II-36, II-41, II-71
Public symbols
 See Symbols, Public
PUTCHAR routine T-15, F-23
Puts() F-24

Q

-q Compiler option I-42
 see compiler, options
QCC_6811 I-45

R

-r Compiler option I-40
-R Compiler option I-43
-r Linker option III-66, III-67
RAM segments I-17, II-44
Rand() F-24
RCODE I-16
Read/write I-10

- Real numbers II-89
- Realloc() F-24
- Recursive J-3
- Reentrant J-3
- Reflne (Assembler listing) II-25, II-108
- Register
 - use of I-11
- Register keyword I-3
- Registers T-24
- Relocatable object (.R07) files II-3, II-11, III-1, III-6, III-25, III-26
- Relocatable segments
 - See Segments, Relocatable
- RMB Assembler directive II-86
- ROM segments I-16, II-44
- Root bank I-9
- RSEG Assembler directive II-78

S

- s Compiler option I-40
- S Compiler option I-45
- S Linker option III-67
- S Assembler option II-12
- Scanf I-19,)
- Segment J-4
- Segments
 - Alignment II-78
 - Alignment III-38
 - Allocation, up/downward III-38, III-74
 - And the 68HC11 architecture II-44
 - Banked III-51
 - Code (ROM) II-44
 - COMMON type II-79
 - COMMON type III-74
 - Defining in Linker (-Z) III-12, III-27
 - Definition II-33
 - Determining size II-62
 - Diagram of example linkage III-28
 - Display list of in a module IV-30
 - In Assembler listings II-24
 - In Assembler programs II-33

- In Linker listings III-37
- INTVEC I-23
- Load address, final III-37
- Origin III-38
- Overlay control III-50
- Overview of II-75
- Relocatable II-76
- Renaming in Librarian IV-34
- RSEG type II-78
- RSEG type III-74
- SHORTAD I-22
- STACK type II-80
- STACK type III-74
- Typing (in Linker) III-73, III-79
- Segments, other names I-17
- SET (Assembler directive) II-83
- Setjmp() F-29
- Short integer
 - length of I-3
- SHORTAD I-17, I-22
- Signed char I-4
- Signed I-49
- Sin() F-29
- Source files
 - See Assembler, Source files
- Source lines
 - See Assembler, Source lines
- Sprintf I-18, F-29
- Sqrt() F-32
- Srand() F-27
- Sscanf I-19, F-28
- STACK Assembler directive II-80
- Stack I-19
 - Parameter passing (banked) I-14
 - passing parameters I-12
- Star II-29
- Startup routine (CSTARTUP) J-4
- Static J-4
- STITL (Assembler directive) II-107
- Strcat() F-30
- Strchr() F-30
- Strcmp() F-30

Strcoll() F-31
Strcpy() F-31
Strcspn() F-31
Strlen() F-32
Strncat() F-32
Strncmp() F-33
Strncpy() F-33
Strpbrk() F-33
Strrchr() F-34
Strspn() F-34
Strstr() F-34
Strtod() F-35
Strtol() F-35
Strtoul() F-36
Structure I-52
Sun host system III-10, III-42
Symbols
 Assembler Declaring II-82
 Case sensitivity III-9
 Defining at link-time III-10, III-42, III-55
 Global/Public II-70, II-84
 Global/public III-30, III-37
 Global/public IV-29
 In Assembler programs II-35
 In Linker listings III-37
 Link-time typechecking III-26, III-30, III-61
 Listing in a library IV-11
 Local II-12, II-25, II-70, II-75
 Local III-64
 Naming II-36
 Pre-defined II-36
 Public II-25, II-27
 Renaming at link-time III-56
 Renaming in Librarian IV-33
 User-defined II-36, II-48, II-82, II-85

T

-T Compiler option I-42
-tn Compiler option I-42
-t Linker option III-68

Tan() F-36
 Target J-4
 TEMP I-17
 Testing T-29
 TITL (Assembler directive) II-107
 Tlower() F-37
 Tools
 debugging T-3
 development T-3
 Toupper() F-37
 Tutorial T-1
 Type check I-2
 Typechecking
 See Symbols, Link-time Typechecking
 Types, data I-50

U

-U I-44
 UBROF T-7, II-3, II-11, II-16, III-1, IV-5, J-4
 UDATA I-17
 Union I-52
 Unix III-10, 42

V

Va_arg() F-28
 Va_end() F-28
 Va_start() F-28
 VAX host system III-10, III-42
 Void I-50
 Volatile I-49

W

-w Compiler option I-39
 -w Linker option III-68
 W Assembler option II-12
 Wait_for_interrupt I-21

Warnings, compilation I-47
WCSTR I-17
Wide listings II-12, II-104
Write, absolute I-10

X

-x Compiler index I-42
-x Linker option III-69
X Assembler option II-108
XCL files
 See Linker, Command (XCL) files
XLIB
 See Librarian
XLINK
 See Linker

Y

-y Compiler option I-41
-Y Linker option III-70

Z

-Z Linker option III-72
 see Linker options
-z Linker option III-71
-z Compiler option I-40
Zero page I-22
ZPAGE Assembler directives II-90
Zpage Compiler keyword I-22
ZVECT segment I-16

